



**Co-Contextual Type Systems:
Contextless Deductive Reasoning for Correct Incremental Type
Checking**

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor rerum naturalium (Dr. rer. nat.)

vorgelegt von

Edlira Kuci, M.Sc.

geboren in Shkodër (Albania).

Referent:	Prof. Dr.-Ing. Mira Mezini
Korreferent:	Prof. Dr. Klaus Ostermann
Korreferent:	Dr. Sebastian Erdweg
Datum der Einreichung:	11. Februar 2019
Datum der mündlichen Prüfung:	25. März 2019

Erscheinungsjahr 2019

Darmstädter Dissertationen

D17

Kuci, Edlira : Co-Contextual Type Systems: Contextless Deductive Reasoning for
Correct Incremental Type Checking
Darmstadt, Technische Universität Darmstadt

Jahr der Veröffentlichung der Dissertation auf TUPrints: 2020
URN: urn:nbn:de:tuda-tuprints-114194
URL: <https://tuprints.ulb.tu-darmstadt.de/id/eprint/11419>

Tag der mündlichen Prüfung: 25.03.2019

Veröffentlicht unter CC BY-SA 4.0 International
<https://creativecommons.org/licenses/>

Acknowledgements

First, I would like to thank my supervisor, Prof. Mira Mezini, for the opportunity to do my PhD and achieve that goal. Thank you for supporting me in my objectives and the research guidance to achieve these objectives. Your feedback has always been useful and put my research forward. Moreover, I am very grateful for your support and understanding beyond my work.

Another person without whom I would not have achieved the successful finalization of my PhD is Sebastian Erdweg. Your guidance strongly influenced the way I think and work. From your example I better understood how to tackle problems and constructively pursue the solutions. I am deeply grateful for your advice and your reliability.

Especially, I want to thank Gudrun Harris for her understanding and making my PhD life easier by facilitating all the paper work. Thank you for supporting my work and finding funds to put my ideas forwards. I am thankful and grateful for always encouraging me with your positive attitude.

I would like to thank all my colleagues of the Software Technology Group for making the years of my PhD more fun. Especially, I want to thank Oliver Bracevac, Sylvia Grewe and Manuel Weiel for the useful feedback and discussions that helped me in my research.

A special thanks goes to my friends Ervina and Khadidja for their support and friendship. I will always be grateful to you for upholding me in the long and sometimes not so easy journey of my PhD. Thank you for always backing me up and for motivating me to see things positively. You had a great influence in me finishing my PhD.

Finally, I want to thank my parents Edvalda and Fatbardh, my sister Sivi, my brother Fatjon and my partner Ralf. Thank you for your unconditional love and support. Thank you for being my strength. I would not have made it so far without you. Faleminderit, ju dua shume dhe ju jam perjete mirenjohese. In particular, thank you Ralf for believing in me, for being patient with me and for being there for me. Thank you for always supporting me in my objectives and helping me to move forward.

Abstract

This thesis proposes a novel way of performing type checking, whose results are incremental, depending on the provided local information. This new way of type checking is called co-contextual, where all context information of expressions, methods, classes, etc., is removed. Instead, we introduce corresponding structures using requirements. Standard type systems are translated to the co-contextual ones systematically using dualism as technique.

Type systems play an important role to prevent execution errors from occurring during runtime. They are used to check programs statically for potential errors. Programs are type checked against a given set of rules. Depending on these rules programs are well-typed or not. The set of these rules is called typing rules. Each type rule associates types to the constructs of a program given a certain context. There can be different forms of contexts, depending on the features of the typed programming language. Functional languages use a typing context of variables and their types; object-oriented (OO) languages use additional class tables. Class tables are used for example to ensure that method and class declarations are well-typed.

Type checking is performed top-down. While traversing the syntax tree of a program, typing contexts are extended with information on the expressions and their types. In case of OO, class tables are extended with clauses from class declarations, including the class members, i.e., fields, methods, or constructors. Contexts are passed through the nodes of the syntax tree in order to coordinate type checking between them. Therefore, while traversing the syntax tree top-down, the type checker creates dependencies between otherwise independent subexpressions. This way, it inhibits incrementalization and parallelization of type checking. That is, a change to a node of the syntax tree would require to redo the type check of the whole syntax tree.

In this thesis a novel formulation of type systems is proposed, in order to remove dependencies between subexpressions. We propose a co-contextual formulation of typing rules that depends only on the local program constructs, e.g., expressions, methods, classes. The co-contextual typing rules have as conclusion a type and sets of *requirements*. That is, contexts and class tables are replaced by the dual concept of context and class table requirements. In addition, operations on contexts and class tables are replaced by new dual operations on requirements. The co-contextual type checker traverses a syntax tree bottom-up and merges context requirements of independently checked subexpressions. We describe a method for systematically constructing a co-contextual formulation of type rules from a regular context-based formulation and we show how co-contextual type rules give rise to incremental type checking.

We derive co-contextual type checkers for functional and OO languages. As a representative of functional languages we consider PCF and extensions of it: records, parametric

polymorphism, structural subtyping and let-polymorphism. Also, we investigate featherweight java (FJ) as the basis of OO languages and extensions of it: method overloading and generics. We build a co-contextual type checker for FJ enabling key features of OO languages: subtype polymorphism, nominal typing and implementation inheritance. The dualism between the co-contextual and contextual type systems preserves the correctness of the contextual calculus. That is, we prove the correctness of the co-contextual calculus via the equivalence between contextual type rules and their co-contextual formulations.

We implemented an incremental type checker for PCF along with a performance evaluation showing that co-contextual type checking has performance comparable to standard context-based type checking, and incrementalization can improve performance significantly. Regarding FJ, we implemented a co-contextual type checker with incrementalization and compared its performance against `javac` on a number of realistic programs. Our performance evaluation shows significant speedups for the co-contextual type checker with incrementalization in comparison to `javac`.

Zusammenfassung

Diese Arbeit schlägt einen neuen Ansatz für Typsysteme vor, in dem die Ergebnisse einer Überprüfung sich inkrementell warten lassen. Die inkrementelle Wartung steht hierbei in Abhängigkeit zu der bereitgestellten lokalen Information. Diese neue Art der Typenüberprüfung wird als ko-kontextuell bezeichnet, wobei alle Kontextinformationen von Ausdrücken, Methoden, Klassen usw. entfernt werden. Stattdessen werden Strukturen eingeführt, welche die entsprechenden Informationen als Bedingungen darstellen. Standard Typsysteme werden systematisch mit Hilfe einer Dualismustechnik in die ko-kontextuellen Systeme übersetzt.

Typsysteme spielen eine wichtige Rolle bei der Prävention von Laufzeitfehlern. Sie werden genutzt um Programme statisch auf potentielle Fehler zu prüfen. Die Typüberprüfung von Programmen wird auf der Basis vorgegebener Regeln des Typsystems durchgeführt. Diese Regeln werden im allgemeinen als Typregeln bezeichnet. Typregeln assoziieren Typen mit den Konstrukten eines gegebenen Programmes und eines dem Programm gemäßen Kontextes. Es gibt verschiedene Arten von Kontexten, die sich aus den Merkmalen der typisierten Programmiersprache ableiten. Funktionale Programmiersprachen nutzen einen Typkontext bestehend aus Variablen und deren Typ; objektorientierte (OO) Sprachen nutzen zusätzliche Klassentabellen als Kontext. Klassentabellen werden zum Beispiel eingesetzt um zu überprüfen, ob Klassen- und Methodendeklarationen wohltypisiert sind.

Die Typenprüfung erfolgt beim Durchlaufen des Syntaxbaums eines Programms von oben nach unten. Hierbei werden die Kontexte um Informationen über die Ausdrücke und deren Typen erweitert. Im Falle von OO werden Klassentabellen um Klauseln aus Klassendeklarationen erweitert, einschließlich deren Klassenmitglieder, d.h. Felder, Methoden oder Konstruktoren. Kontexte werden durch die Knoten des Syntaxbaums gereicht, um die Typüberprüfung zwischen ihnen zu koordinieren. Daher erzeugt die Typüberprüfung beim Durchlaufen des Syntaxbaums von oben nach unten Abhängigkeiten zwischen ansonsten unabhängigen Teilausdrücken. Diese Art der Typüberprüfung verhindert die Inkrementalisierung und Parallelisierung der Prüfung. Das heißt, eine Änderung an einem Knoten des Syntaxbaums würde erfordern, die Typprüfung des gesamten Syntaxbaums erneut durchzuführen.

Diese Arbeit schlägt eine neue Formulierung bestehender Typsysteme vor, um Abhängigkeiten zwischen Teilausdrücken zu entfernen. Wir stellen eine ko-kontextuelle Formulierung von Typregeln vor, die sich nur auf lokale Programmkonstrukte bezieht, zum Beispiel auf Ausdrücke, Methoden oder Klassen. Die ko-kontextuellen Typregeln haben als Schlussfolgerung einen Typ und Mengen von Bedingungen. Das heißt, Typkontext und Klassentabellen werden durch das duale Konzept von Kontext- und Klassentabellenbedingungen ersetzt. Außerdem, werden Operationen, die auf dem Typkontext

und den Klassentabellen definiert sind, durch entsprechende duale Operationen ersetzt. Die ko-kontextuelle Typüberprüfung durchläuft den Syntaxbaum von unten nach oben und führt die Kontextbedingungen von unabhängigen Teilausdrücken zusammen. Wir Beschreiben eine systematische Methode für die Konstruktion von ko-kontextuellen Typregeln ausgehend von einer kontextbasierten Formulierung der Typregeln und zeigen wie ko-kontextuelle Typregeln die Inkrementalisierung der Typüberprüfung nutzbar machen.

Wir überführen kontextbasierte Typüberprüfung in eine ko-kontextuelle Formulierung für Funktionale- und OO-Sprachen. Als Vertreter funktionaler Sprachen betrachten wir PCF und seine Erweiterungen: Records, parametrischer Polymorphismus, strukturelle Subtypisierung und Let-Polymorphismus. Außerdem untersuchen wir Featherweight-Java (FJ) als Grundlage für OO-Sprachen und deren Erweiterungen: Methodenüberladung und Generics. Wir entwickeln eine ko-kontextuellen Typüberprüfung für FJ, welche die wichtigsten Merkmale von OO-Sprachen ermöglicht: Subtyp-Polymorphismus, nominale Typisierung und Vererbung von Implementierungen. Der Dualismus zwischen dem ko-kontextuellen und dem kontextbasierten System erhält die Korrektheit der kontextbasierten Formulierung. Das heißt, wir beweisen die Korrektheit der ko-kontextuellen Formulierung durch die Äquivalenz zwischen kontextbasierten Typregeln und ihren ko-kontextuellen Entsprechungen.

Wir haben eine ko-kontextuelle inkrementelle Typüberprüfung für PCF implementiert und stellen diese zusammen mit einer Leistungsbewertung vor, die zeigt, dass die gesamte ko-kontextuelle Typprüfung eine vergleichbare Leistung mit der standardmäßige kontextbasierte Typprüfung aufweist und die Inkrementalisierung die Leistung deutlich verbessern kann. Bezüglich FJ, haben wir eine ko-kontextuelle inkrementelle Typüberprüfung implementiert und deren Leistung mit `javac` in einer Reihe von realistischen Programmen verglichen. Unsere Leistungsbewertung zeigt eine signifikante Verbesserung der Geschwindigkeit für ko-kontextuelle inkrementelle Typüberprüfung im Vergleich zu `javac`.

Contents

I. Introduction	13
1.1. Contributions of This Thesis	20
1.2. Publications	21
1.3. Structure of This Thesis	23
 II. Co-contextual Type Checkers for Functional Languages	 25
Part II Overview	27
 2. Background and Motivation	 29
2.1. PCF: Syntax and Typing Rules	29
2.2. Contextual and Co-Contextual PCF by Example	30
 3. Co-contextual PCF	 33
3.1. Constructing Co-Contextual Type Systems	33
3.1.1. Co-Contextual Syntax and Operations	34
3.1.2. Co-Contextual PCF	36
3.2. Incremental Type Checking	37
3.2.1. Basic Incrementalization Scheme	38
3.2.2. Incremental Constraint Solving	39
3.2.3. Eager Substitution	40
3.3. Technical Realization	41
3.4. Performance Evaluation	42
 4. Simple Extensions of PCF	 47
4.1. Records	47
4.2. Parametric Polymorphism	48
4.3. Structural Subtyping	50
 5. Co-contextual Let-Polymorphism	 55
5.1. Contextual Let-Polymorphism	55
5.2. Co-Contextual Structures for Let-Polymorphism	56
5.3. Co-Contextual Constraints for Let-Polymorphism	57
5.3.1. Partial Generalization Constraint	58
5.3.2. Instantiation Constraint	59
5.3.3. Continuous Solving of Constraints	61
5.4. Co-Contextual Let-Polymorphism by Example	61

5.5. Co-Contextual Typing Rules	62
5.6. Implementation and Evaluation	64
6. Related Work	67
7. Part Summary	71
 III. Co-Contextual Type Checkers for Object-Oriented Languages	 73
Part III Overview	75
 8. Background and Motivation	 77
8.1. Featherweight Java: Syntax and Typing Rules	77
8.2. Contextual and Co-Contextual Featherweight Java by Example	79
 9. Co-contextual Featherweight Java	 83
9.1. Co-Contextual Structures for Featherweight Java	83
9.1.1. Class Types and Constraints	83
9.1.2. Context Requirements	84
9.1.3. Structure of Class Tables and Class Table Requirements	84
9.1.4. Operations on Class Tables and Requirements	85
9.1.5. Class Table Construction and Requirements Removal	87
9.2. Co-Contextual Featherweight Java Typing Rules	90
9.2.1. Expression Typing	92
9.2.2. Method Typing	92
9.2.3. Class Typing	93
9.2.4. Program Typing	93
9.3. Typing Equivalence	94
9.4. Efficient Incremental FJ Type Checking	96
9.5. Performance Evaluation	99
9.5.1. Evaluation on synthesized FJ programs	99
9.5.2. Evaluation on real Java program	101
 10. Co-Contextual Featherweight Java with Generics	 103
10.1. Featherweight Java with Generics	103
10.1.1. Featherweight Java with Generics Typing Rules	105
10.2. Co-Contextual Structures for Featherweight Java with Generics	107
10.2.1. Co-contextual Constraints	107
10.2.2. Bounded requirements	108
10.2.3. Co-Contextual Well-Formedness Rules	109
10.3. Operations on Class tables Requirements	110
10.3.1. Merge Operation	110
10.3.2. Remove Operation	111
10.4. Co-Contextual Featherweight Java with Generics Typing Rules	113

10.5. Summary	116
11. Co-Contextual Featherweight Java with Method Overloading	117
11.1. Featherweight Java with Method Overloading	117
11.1.1. Featherweight Java with Method Overloading: Types, Syntax and Minimal Selection	117
11.1.2. Featherweight Java with Method Overloading Typing Rules	119
11.2. Co-Contextual Structures for Featherweight Java with Method Overloading	121
11.2.1. Co-contextual Constraint	121
11.2.2. Structure of Class Table Requirements	121
11.3. Operations on Class Table Requirements	122
11.3.1. Merge Operation	122
11.3.2. Remove Operation and Minimal Selection of the Required Method	123
11.3.3. Remove Operation with Overloading by Example	125
11.4. Co-Contextual FJ with Method Overloading Typing Rules.	126
11.5. Impact of Co-contextual FJ with Method Overloading on Efficient Incremental Type Checkers	127
12. Related Work	129
13. Part Summary	133
 IV. Conclusions and Future directions	 137
14. Conclusions and Future directions	139
14.1. Conclusions	139
14.2. Future Directions	142
 Bibliography	 145
 Appendix	 151
A. Equivalence of Contextual and Co-Contextual PCF	153
B. Equivalence of Contextual and Co-Contextual FJ	159
B.1. Auxiliary definitions; merge, add, remove	159
B.2. Equivalence of Contextual and Co-Contextual FJ	162

Part I.

Introduction

Type checkers of modern programming languages play an important role in assuring software quality as they try to statically prove increasingly strong invariants over programs. Type checkers operate based on a predefined type system. A type system is formed from a given set of typing rules. These rules are applied by the type system to determine if a program is well-typed and what type the expressions used in a program have. For example, the type checker will indicate that a program is not well-typed when one tries to access a field that it is not declared in the receiver class, or to add two numbers one being integer and the other one being float. Also, it can detect deeper errors. For example, when one wants to change the definition of a data structure, then he does not have to check all files manually because all errors will be available once type checking is performed. In this case, type checking is considered as a maintenance tool. Also, a type system ensures the language safety. Namely, a language that protects its own abstractions. A type checker is part of the compiler and is triggered automatically at compile time. Hence, the programmer does not need to start it additionally.

Type systems involve a broad research field, starting from early 1900. A very well-known contribution in this field is the work from Church in 1940 [Chu40] and later from Curry et al. [CF58] on simply typed lambda calculus (STLC), which is nowadays a basis for functional languages. Later in 1960s type systems were introduced for first languages like Algol by Naur et al. [BBG⁺63]. Continuing in 1970s with more languages like Fortran by Backus [Bac81], Pascal by Wirth [Wir71], System F , F^ω by Girard [Gir71], polymorphic type inference by Milner [Mil], and later by Damas and Milner [DM82], ML by Gordon et al. [GMW79]. In 1990s additional concepts were introduced on higher-order subtyping by Cardelli [Car88a, Car88b], and object-calculus by Abadi and Cardelli [ACV96], etc. Nowadays research is done on complex type systems, like Featherweight Java (FJ) a base type system for object-oriented languages by Igarashi et al. [IK01], or path-dependent types by Amin et al. [ARO14].

Let us briefly describe how the typing rules for STLC look like. Typing rules consist of derivation typing judgements that have the form:

$$\Gamma \vdash e : T$$

Where, Γ is the typing context, e the expression to be type checked, and T its associated type. That is, expressions are type checked under a given context and give as a result a certain type. Context Γ is a set of bindings from variables to their types. Expressions can be variables (x), lambda abstractions ($\lambda x : T. e$), applications ($e e$), or others in more complex type systems. For example, the typing rule for an application is presented below

$$\frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1 e_2 : T_2}$$

The judgements above the line are the premises that must be fulfilled for the rule to be applied, yielding the conclusion: the judgement below the line. This is interpreted: if the expressions e_1 and e_2 have types $T_1 \rightarrow T_2$ and T_1 , correspondingly, under the context Γ , the expression $e_1 e_2$ has type T_2 and context Γ . e_1 and e_2 can be considered as

subexpressions of the application. Hence, we could consider expressions as nodes of a syntax tree and branches of these nodes are their subexpressions.

There are more complex type systems than STLC, that do not type check only expressions, or use only typing contexts while type checking. For example, FJ is a more complex type system than STLC. It uses class tables in addition to the typing context while type checking. That is, FJ typing rules use as part of their judgements the typing contexts and class tables. The latter one is a kind of typing context but it has a more complex structure than the typing context Γ . Class tables do not consist of bindings from variables to types, but they are built from clauses of declared classes and their members like fields, methods, etc. Typing rules for expressions in FJ are different from the ones in STLC and more complex because they involve field access, method invocation, etc. For example, the judgement for field access is given below:

$$\Gamma; CT \vdash e.f : T$$

CT stands for class tables and is part of the judgement in addition to the context Γ , where f is the accessed field and T its type. This type is retrieved from the class table, by searching for a declaration of f in the type of e or its parent classes. There are additional typing rules for methods, or class declarations. These typing rules do not yield types as conclusions, but they indicate new declarations of a class or method to be added to the class table. That is, they ensure that the new declarations are well-typed with respect to the rest of the declarations already part of the class table. In the following of this thesis we will talk more about FJ and its features.

Typically, a type checker starts processing at the root node of a syntax tree and takes a typing context as an additional input to coordinate between sub-derivations by assigning types to jointly used variables. While traversing down the tree, type checkers extend the context with type bindings, making them available in subexpressions. When the type checker reaches a variable, it looks up the corresponding binding in the typing context. Since variables constitute leaves of the syntax tree, the downward traversal ends. The type checker then propagates the derived types back upward the syntax tree. In summary, while types flow bottom-up, typing contexts flow top-down – overall, the type system runs in a "down-up" mode.

The construction of current type checkers inherently deals with dependencies in subexpressions, by re-typing the whole program. In essence this behavior is a consequence of the down-up nature of the current checkers because the typing of subexpressions depends on the typing of parent expressions (to retrieve the context) and vice versa (to retrieve the types). This means that current type checkers are limited w.r.t., incremental, parallel, and compositional type checking. Consequently, type checking can take considerable time due to a) size of software systems and b) complexity of the type system. The more complex the typing rules are the longer it takes type checking. Efficient type checking in the presence of large software systems is of great interest to industry. In addition, the connection between type checking and the complexity of the language hampers the development of more complex type systems because they will not be used in practice if the checking takes too long.

This thesis describes a novel technique of performing type checking. We present a generic method for constructing new type systems from standard type systems that eliminates the expression dependencies, inherently needed by the construction of the standard type systems. The method used to realize this translation is *dualism*. To remove the dependencies between subexpressions, we propose to eliminate all top-down propagated contexts. We replace them by the dual concept of bottom-up propagated *requirements*. This enables bottom-up type checking, which starts at the leaves of a syntax tree and gradually deduces additional type information. We call such a type checking *co-contextual*. Whereas, the traditional type checking is called *contextual*. A program is well-typed in co-contextual setting if all requirements are satisfied. Namely, the requirements sets are empty. The dualism allows us to construct the co-contextual type systems from the contextual, by systematically translating the typing rules. The correctness of the co-contextual type system is proven via the equivalence between the contextual and co-contextual typing rules. We explore representative type systems of functional and object-oriented languages, and apply our method to obtain the corresponding co-contextual type systems. Moreover, we show that co-contextual type checkers can be used for fine-grained incrementalization because they remove dependencies between subexpressions. That is, co-contextual formulation of type rules enables incrementalization at the level of type checking.

We consider Programmable Computable Functions (PCF) as a representative of functional languages because it is an extended version of STLC and a simplified version of modern programming languages like ML, Haskell, Lisp, etc. As described above, PCF uses typing context during type checking. In order to construct a co-contextual type checker, we apply our method and remove the typing context. Instead, we introduce the dual concept of bottom-up propagated *context requirements*. The contextual formulation is based on a typing context and operations for looking up, splitting, and extending the context. The co-contextual formulation replaces the operations on the typing context with the dual operations of generating, merging, and satisfying context requirements. Whenever a traditional type checker would look up variable types in the typing context, the bottom-up co-contextual type checker generates fresh type variables and generates context requirements stating that these type variables need to be bound to actual types; it merges and satisfies these requirements as it visits the syntax tree upwards to the root. As an example, we consider the following simple lambda expression:

$$\lambda x : Num. x + x$$

Note that here and throughout this thesis we use metavariables U to denote unification variables as placeholders for actual types. The co-contextual type checker generates fresh unification variables independently for both variables: $x : U_0$ and $x : U_1$. Moving up the syntax tree these two requirements can be merged deducing that U_0 and U_1 should be equal since they represent the types of the same variable. Next, we take into consideration the lambda abstraction λ . The context is extended with the actual type of the variable x . The dual to extending the context is removing the requirements corresponding to x because now we know its actual type. This allows us to resolve the unification variables

bound to x and replace them by the actual type (Num) of the variable x . Therefore, all requirements for x are satisfied, and the requirement set is empty. Moreover, we know that the $+$ operator takes two numbers Num as operands. As a result, the above expression is well-typed because the type Num of the annotated variable x is the same as the type of the $+$ operands. If x in the lambda abstraction would be annotated to *String*, then the expression is not well-typed. That is, type checker throws a typing error because numbers are different from strings. Enabling a co-contextual type checking with incrementalization for PCF is a good basis for almost all functional languages.

In this thesis, in addition to functional languages, we investigate co-contextual formulation of type systems for statically typed object-oriented (OO) languages, the state-of-the-art programming technology for large-scale software systems. We use Featherweight Java [IK01] (FJ) as a representative calculus for these languages. This is paving the way to efficient incremental type checkers for OO languages. Constructing a co-contextual type system yields a novel formulation of Igarashi et al.’s Featherweight Java (FJ) type system. In order to translate the original FJ to a co-contextual formulation of it, we have to remove all dependencies between the language constructs. We observe that the general principle of removing and replacing the typing context and its operations with co-contextual duals carries over to the *class table*. The latter is propagated top-down and completely specifies the available classes in the program, e.g., member signatures and superclasses. Dually, a co-contextual type checker propagates *class table requirements* bottom-up. This data structure specifies requirements on classes and members and accompanying operations for generating, merging, and removing these requirements. These operations on class table requirements are dual to the operation on the class table. For example, declaring a member of a class, i.e., adding this member to the class table, is dual to removing corresponding requirements of this member.

However, defining appropriate merge and remove operations on co-contextual class table requirements poses significant challenges, as they substantially differ from the equivalent operations on context requirements. Context requirements are derived from the language features of PCF, such as the global namespace and structural typing. In contrast, class table requirements are derived from the much more complex language features of FJ such as context dependent member signatures (subtype polymorphism), a declared type hierarchy (nominal typing), and inherited definitions (implementation inheritance).

For an intuition of class table requirements and the specific challenges concerning their operations, consider the example below:

```
new List().add(1).size()+new LinkedList().add(2).size();
```

(R_1) List.init()	(R_4) LinkedList.init()
(R_2) List.add : Int \rightarrow U_1	(R_5) LinkedList.add : Int \rightarrow U_2
(R_3) $U_1.size : () \rightarrow U_3$	(R_6) $U_2.size : () \rightarrow U_4$

Type checking the operands of $+$ yields the class table requirements R_1 to R_6 . As for PCF, we use the metavariables U as placeholders for actual types. For example, the

invocation of method *add* on `new List()` yields a class table requirement R_2 . The goal of co-contextual type checking is to avoid using any context information, hence we cannot look up the signature of `List.add` in the class table. Instead, we use a placeholder U_1 until we discover the definition of `List.add` later on. As consequence, we lack knowledge about the receiver type of any subsequent method call, such as *size* in our example. This leads to requirement R_3 , which states that a (yet unknown) class U_1 should exist that has a method *size* with no arguments and a (yet unknown) return type U_3 . Assuming $+$ operates on integers, type checking the $+$ operator later unifies U_3 and U_4 with `Int`, thus refining the class table requirements.

To illustrate issues with merging requirements, consider the requirements R_3 and R_6 regarding *size*. Due to nominal typing, the signature of this method depends on the existence of classes U_1 and U_2 , where it is yet unknown how these classes are related to each other. It might be that U_1 and U_2 refer to the same class, which implies that these two requirements overlap and the corresponding types of *size* in R_3 and R_6 are unified. Alternatively, it might be the case that U_1 and U_2 are distinct classes, individually declaring a method *size*. Unifying the types of *size* from R_3 and R_6 would be wrong. Therefore, it is locally indeterminate whether a merge should unify or keep the requirements separate.

To illustrate issues with removing class requirements, consider the requirement R_5 . Suppose that we encounter a declaration of *add* in `LinkedList`. Just removing R_5 is not sufficient because we do not know whether `LinkedList` overrides *add* of a yet unknown superclass U , or not. Again, the situation is locally indeterminate. In case of overriding, FJ requires that the signatures of overriding and overridden methods to be identical. Hence, it would be necessary to add constraints equating the two signatures. However, it is equally possible that `LinkedList.add` overrides nothing, so that no additional constraints are necessary. If, however, `LinkedList` inherits *add* from `List` without overriding it, we need to record the inheritance relation between these two classes, in order to be able to replace U_2 with the actual return type of *size*.

The example illustrates that a co-contextual formulation for nominal typing with subtype polymorphism and implementation inheritance poses challenging research questions, which we will address in this thesis.

To summarize, since co-contextual type checkers do not coordinate subderivations, it is possible to incrementalize them systematically by applying memoization, incremental constraint solving, and eager substitution. Such incrementalization is independent of the module structure of the program.

1.1. Contributions of This Thesis

In this thesis, we present a generic method for systematically constructing co-contextual type systems from the traditional context-based type systems. This is a novel approach to enable incrementalization at the level of type checking. This is realized by removing contexts and all dependencies that come from their usage. We describe how co-contextual type systems give rise to incremental type checking, given that the type rules are in algorithmic form.

As part of this thesis, we construct co-contextual type systems for a large set of language features. We derive the initial ideas for co-contextual type system construction from studying the translation of the contextual PCF type system. Building on these ideas, we extend co-contextual PCF with additional language features, e.g., parametric polymorphism and let-polymorphism. These extensions are an important step towards a co-contextual type system for a fully-fledged functional language, such as Haskell. Furthermore, we apply our technique to support co-contextual type checkers for object-oriented languages. In this contribution, we construct the co-contextual type systems for FJ and extensions of FJ. We provide correctness proofs for co-contextual type systems of PCF and FJ, based on proofs of equivalence to the corresponding contextual type systems. Finally, we describe the implementation of efficient incremental co-contextual type checkers for PCF, FJ and their extensions.

Co-Contextual Type Checkers for PCF and Extensions of PCF.

Co-contextual type system for PCF is a novel way of performing type checking introduced by Erdweg et al. [EBK⁺15]. Dualism is used as technique to translate the traditional PCF to the co-contextual one. The typing context is removed, therefore the dependencies that come from its usage during type checking are removed. The typing context is replaced by the dual structure of context requirements. The operations on context requirements are dual to the operations on typing contexts.

The technique of dualism for the base PCF type system was conceived together with Erdweg. As part of this thesis, we extend PCF with records, parametric polymorphism, structural subtyping and let-polymorphism. We construct co-contextual type systems for each of these extensions. Also, we describe the required changes systematically applied to co-contextual PCF in order to support its extensions. However, the difficulty to enable the translation from contextual to co-contextual type systems is increased depending on the language constructs that are added to PCF. For example, co-contextualizing let-polymorphism is challenging, since variables can be of a ground type or a polymorphic type. In the co-contextual setting the concrete knowledge about these variables is obtained only after correlating their declaration and usage.

Co-Contextual Type Checkers for FJ and Extensions of FJ.

We consider two research questions: (a) Can we formulate an equivalent co-contextual type system for FJ by duality to the traditional formulation, and (b) if yes, how to define an incremental type checker based on it with significant speedups? Addressing these questions is an important step towards a general theory of incremental type checkers for statically typed OO languages, such as Java, C#, or Eiffel. In this thesis, we show that it is feasible

to construct a co-contextual formulation of FJ's type system by duality to the traditional type system formulation by Igarashi et al. [IK01]. Our formulation replaces the class table by its dual concept of class table requirements and replaces field/method lookups, class table duplication, and class table extension by the dual operations of requirements generation, merging, and removing. In particular, answering the above question and defining the semantics of merging and removing class table requirements in the presence of nominal types, OO subtype polymorphism, and implementation inheritance constitute a key contribution of this thesis. After describing how to build a co-contextual type checker for FJ, we consider two extension of FJ: generics and method overloading. We translate generics and method overloading type systems to corresponding co-contextual ones and elaborate the difficulties that these translations pose.

Proof of Equivalence Between Contextual and Co-Contextual Type Checkers for PCF, FJ

We provide a proof of equivalence between contextual and co-contextual PCF and FJ. Induction and other techniques are used to prove that the translation is correct. Namely, contextual and co-contextual type checkers yield the same results. For example, if an expression is well-typed in the contextual setting and has a certain type T , then it is well-typed also in the co-contextual setting having the same type T and the requirements set is empty. The same holds when a program is not well-typed, i.e., if a program has a typing error in the contextual type checking then it should have the same typing error in the co-contextual one.

Incremental Type Checkers for PCF and FJ, and Performance Evaluations.

Co-contextual type checking gives rise to incremental type checking. We describe how to implement efficient incremental type checking including optimizations for co-contextual PCF and FJ. We compare the non-incremental and incremental performance of the co-contextual type checkers to the context based ones. The performance evaluation shows considerable speedups for the incremental type checking. With respect to FJ, we evaluate the initial and incremental performance of the co-contextual FJ type checker on synthesized FJ programs and realistic java programs by comparison to javac and a context-based implementation of FJ.

The author in collaboration with others has previously published many of these contributions in the proceedings of international conferences and workshops.

1.2. Publications

The following publications were created in the context of the research performed for this thesis with the description of the author's contribution:

1. Sebastian Erdweg, Oliver Bračevac, Edlira Kuci, Matthias Krebs, Mira Mezini: A co-contextual formulation of type rules and its application to incremental type checking. *In Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2015.
Author's contribution: The material in this paper is in account of many discussions

to materialize the initial idea of co-contextual type checking for PCF. Concretely, I dealt with the technical part of records and parametric polymorphism and the proof of the equivalence theorem between contextual and co-contextual PCF.

2. Edlira Kuci, Sebastian Erdweg, Mira Mezini: Toward incremental type checking for Java. *In Companion Proceedings of SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH). 2015.*

Author's contribution: The material of this paper is in account of my vision to extend co-contextual type checkers to support full languages, like Java.

3. Edlira Kuci, Sebastian Erdweg, Oliver Bračevac, Andi Bejleri, Mira Mezini: A Co-contextual Type Checker for Featherweight Java. *In Proceedings of European Conference on Object-Oriented Programming (ECOOP). 2017.*

Author's contribution: The material of this paper is in account of my work to apply co-contextual type systems to OO languages and the actual construction of a co-contextual type system that supports the key features of OO languages.

4. Tamás Szabó, Edlira Kuci, Matthijs Bijman, Mira Mezini, and Sebastian Erdweg. Incremental Overload Resolution in Object-Oriented Programming Languages. *In Proceedings of International Workshop on Formal Techniques for Java-like Programs (FTfJP). ACM, 2018.*

Author's contribution: The material of this paper is in account of insights that came out of the earlier problems we had seen in co-contextual method overloading. Concretely, I described the co-contextual setting.

1.3. Structure of This Thesis

The remainder of this thesis is structured as follows.

Part II - Co-contextual Type Checkers for Functional Languages

describes the translation from contextual to co-contextual type checkers for PCF and extensions of it. Chapter 3 gives a description of the context requirements, their operations, and how the co-contextual typing rules for PCF are constructed. Then, we give the theorem of equivalence between contextual and co-contextual type systems. We have a detailed proof of it in the Appendix A. Next, we show implementation details and how to obtain an incremental type checker from the co-contextual one. Finally, we show our performance evaluation and the speedups gained from the incrementalized co-contextual type checker. Chapter 4 gives a description of co-contextual type systems for extension of PCF. We translate from the traditional type checkers to co-contextual ones for records, parametric polymorphism and structural subtyping. Chapter 5 focuses on let-polymorphism. In this chapter, we describe the translation from contextual to a co-contextual type checker supporting let-polymorphism and we discuss the difficulties that arise from this translation. Then, we describe the changes required in constraints with respect to the co-contextual PCF in order to feature let-polymorphism, and show the co-contextual typing rules. Finally, we present the technical realization, where we describe the implementation of co-contextual PCF with let-polymorphism. Chapter 6 gives an overview of related work.

Part III - Co-contextual Type Checkers for Object-Oriented Languages

presents co-contextual type checkers for FJ and extensions of it. Chapter 9 introduces the notion of class table requirements and operations on them that incorporate the key features of FJ such as subtype polymorphism, nominal typing and inheritance. Then, we describe the co-contextual typing rules for FJ. Next, we give the theorems required to prove typing equivalence between contextual and co-contextual FJ type systems. We have a detailed proof of them in the Appendix B. Finally, we describe implementations details and optimizations performed for the co-contextual type checker. Also, we present a performance evaluation realized on synthesized FJ programs and on real Java programs, including a large number of classes. In our evaluation we compare the performances of context-based FJ and javac to non-incremental and incremental co-contextual FJ. Chapter 10 presents the systematic changes required to be done to the co-contextual FJ in order to feature generics. Since generics is a new language construct added to FJ, we need to change the types, constraints, and requirements sets of co-contextual FJ. Furthermore, we change the merge and remove operations on class table requirements, so they support generic methods and classes. Chapter 11 describes method overloading in FJ and why adding overloading to co-contextual FJ is challenging. Then, we give a detailed description of the operations on method requirements, which are the only requirements affected by method overloading. Next, we describe the construction of co-contextual typing rules for method overloading. Finally, we discuss the impact of method overloading on the efficiency of the incremental co-contextual type checker. Chapter 12 gives an overview of related work.

Part IV - Conclusion and Future Directions

concludes this thesis and discusses directions for future work. We elaborate applications to more complex type systems than the ones presented in this thesis and enhancements to the presented technique. Finally, we suggest applications of our technique to improve type checking in the areas of concurrent and distributed systems.

Part II.

Co-contextual Type Checkers for Functional Languages

Part II Overview

In this part, we describe how to construct co-contextual type checkers for functional languages from their contextual counterparts. We consider PCF as a representative of functional languages and apply our method to the constraint-based type system of PCF. The constraint-based version of the PCF type system is used for type inference. We provide a description of the type system below to establish the respective types, constraints and typing rules. In the following chapters, we give a detailed description of the translation to co-contextual type system. We present the context requirements and detail the operations on requirements required for functional languages. Furthermore, we show how the co-contextual type checker gives rise to incrementalization. We describe respective techniques for continuous solving of constraints and memoization and evaluate the performance of the incrementalization in contrast to the contextual. To illustrate the systematic translation to a co-contextual type system, we consider extensions of PCF: records, System F, structural subtyping, and let-polymorphism. We describe the systematic changes done to co-contextual PCF in order to support these extensions.

2. Background and Motivation

In the following two sections, we present the syntax and typing rules for PCF. Then, we give an example to illustrate how contextual and co-contextual PCF type checkers work.

2.1. PCF: Syntax and Typing Rules

PCF is an extended version of the typed lambda calculus. Below, we give the syntax for expressions, types, and contexts of PCF:

$$\begin{array}{ll}
 e ::= n \mid x \mid \lambda x:T. e \mid \text{fix } e & \text{expressions} \\
 \quad \mid ee \mid e + e \mid \text{if0 } e e e & \\
 T ::= \text{Num} \mid T \rightarrow T & \text{types} \\
 \Gamma ::= \emptyset \mid \Gamma; x:T & \text{typing contexts}
 \end{array}$$

The expressions can be numbers, variables, lambda abstraction, fix point, application, $+$ and if0 . The types are only numbers Num and function types $T \rightarrow T$. Γ is the typing context, where the expression e is type checked. The typing context is a mapping from variables to their corresponding types, which could be empty, or extended with more variable bindings.

Reformulating type rules such that they produce type constraints instead of performing the actual type check is a standard technique in the context of type inference and type reconstruction [Pie02]. As stated above, we assume that the original type rules are given in a constraint-based style. That is, we assume the original typing judgment has the form $\Gamma \vdash e : T \mid C$, where T is the type of e if all type constraints in set C hold.

In constraint-based PCF, type constraints take the form of equalities:

$$c \in C ::= T = T \quad \text{type constraints}$$

Figure 2.1 shows the constraint-based contextual type rules of PCF. The rules T-Num, T-Var and T-Abs are straightforward. The rule T-Add describes the $+$ operator and represents the addition of two numbers Num , indicating that both expressions e_1 and e_2 should be of type Num , which is ensured via the equality constraints $T_1 = \text{Num}$ and $T_2 = \text{Num}$. These constraints are added to the resulting constraints set. The rules T-App and T-Fix are also straightforward. The rule T-If0 has the expression e_1 , which represents the condition of if0 . The type of e_1 , which is compared to the number zero , which is a Num . The expressions e_2 and e_3 represent the cases of if0 . Both expressions should have the same type ($T_2 = T_3$), which is also the resulting type of the expression if0 .

2. Background and Motivation

$$\begin{array}{c}
\text{T-Num} \frac{}{\Gamma \vdash n : \text{Num} \mid \emptyset} \qquad \text{T-Var} \frac{\Gamma(x) = T}{\Gamma \vdash x : T \mid \emptyset} \\
\\
\text{T-Abs} \frac{\Gamma; x : T_1 \vdash e : T_2 \mid C}{\Gamma \vdash \lambda x : T_1. e : T_1 \rightarrow T_2 \mid C} \qquad \text{T-Fix} \frac{\Gamma \vdash e : T \mid C \quad U \text{ is fresh}}{\Gamma \vdash \text{fix } e : U \mid C \cup \{T = U \rightarrow U\}} \\
\\
\text{T-Add} \frac{\Gamma \vdash e_1 : T_1 \mid C_1 \quad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 + e_2 : \text{Num} \mid C_1 \cup C_2 \cup \{T_1 = \text{Num}, T_2 = \text{Num}\}} \\
\\
\text{T-App} \frac{\Gamma \vdash e_1 : T_1 \mid C_1 \quad \Gamma \vdash e_2 : T_2 \mid C_2 \quad U \text{ is fresh}}{\Gamma \vdash e_1 e_2 : U \mid C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow U\}} \\
\\
\text{T-If0} \frac{\Gamma \vdash e_1 : T_1 \mid C_1 \quad \Gamma \vdash e_2 : T_2 \mid C_2 \quad \Gamma \vdash e_3 : T_3 \mid C_3}{\Gamma \vdash \text{if0 } e_1 e_2 e_3 : T_2 \mid C_1 \cup C_2 \cup C_3 \cup \{T_1 = \text{Num}, T_2 = T_3\}}
\end{array}$$

Figure 2.1.: A contextual constraint-based formulation of the type system of PCF.

2.2. Contextual and Co-Contextual PCF by Example

In this section, we give an example to compare the contextual and co-contextual type checkers. This example gives an intuition of the new way of type checking that we propose in this thesis. Moreover, we explain via this example how the co-contextual type checker removes the dependencies.

We consider the process of type checking the simply-typed expression $\lambda f : \alpha \rightarrow \text{Num}. \lambda x : \alpha. f x + f x$ (α is arbitrary but fixed type) with respect to a contextual and a co-contextual type checkers.

Figure 2.2 depicts the contextual type checking, which shows the syntax tree of the expression explicitly, marking the application terms with **app**. We attach typing contexts and types to syntax tree nodes on their left-hand and right-hand side, respectively. The type attached to a node represents the type of the whole subexpression. Moving down the syntax tree context Γ is extended with information for variables f and x because the nodes are lambda abstractions and give the actual types of f and x . Then, the type checker encounters the $+$ operator, which takes as arguments two numbers Num and returns a number. Therefore, the types of the two **app** nodes are Num . Moving down the syntax tree the information about f and x is passed to the leaves of the syntax tree and used to type check the two f and x variables. As shown in Figure 2.2 the typing context flows top-down and coordinates the type checking between different subexpression.

Figure 2.3 depicts the process of type checking the same expression by a co-contextual type checker. We separate a node's type T and context requirements R by a vertical

2.2. Contextual and Co-Contextual PCF by Example

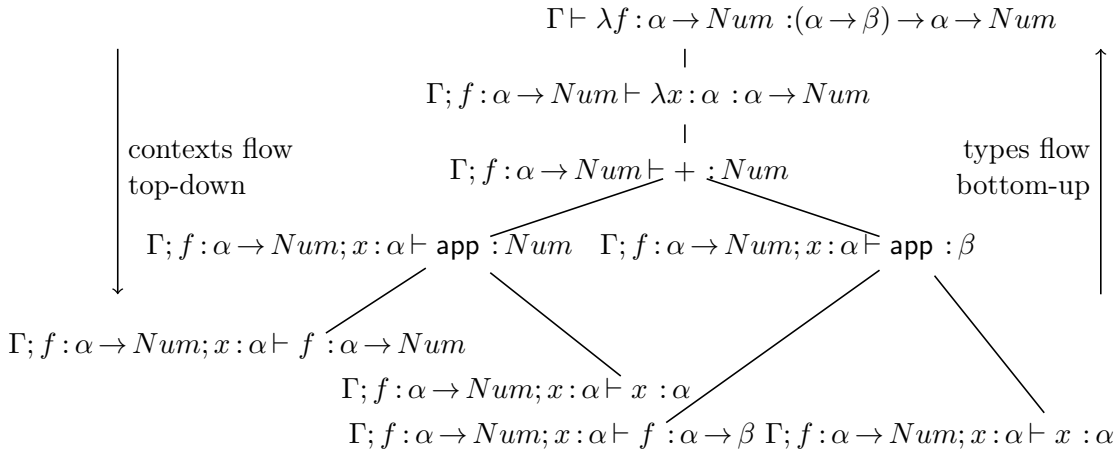


Figure 2.2.: Contextual type checking propagates contexts top-down.

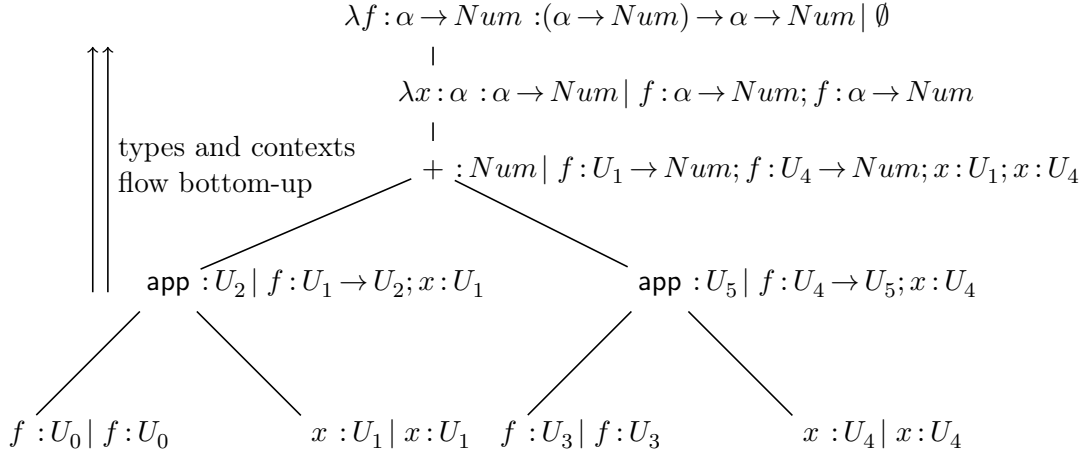


Figure 2.3.: Co-contextual type checking propagates contexts bottom-up.

bar $T \mid R$. The following tree illustrates how a co-contextual type checker enables type checking a program bottom-up.

The co-contextual type checker starts at the leaves of the syntax tree without using the typing context. Namely, the type checker has no information about the actual types of the variables f and x . Therefore, fresh *unification variables* (U) are generated as placeholders for their actual types. Each occurrence of f and x generates a fresh unification variable, hence, there is no coordination between the different leaves of the syntax tree.

In the next step, the type checker encounters the application terms. The type checker refines the types U_0 and U_3 to function types; $U_1 \rightarrow U_2$ and $U_4 \rightarrow U_5$, respectively, which become the result type of the application terms. Next, type checker

2. Background and Motivation

collects all context requirements of subexpressions and applies the type substitutions $\{U_0 \mapsto U_1 \rightarrow U_2, U_3 \mapsto U_4 \rightarrow U_5\}$ to them, and propagates them upward. This changes the type of f in the requirement set.

Next, the type checker encounters the $+$ operator. We know from the T_{ADD} typing rule that its subexpressions and the resulting types are Num . Therefore, the types of **app** subexpressions are refined to Num and the type of f is further refined, changing the types of f in the requirements to $U_1 \rightarrow Num$ and $U_4 \rightarrow Num$.

When the λ -abstraction on x is reached, the type checker knows the actual type of the variable x , which is α . Therefore, all requirements on that variable are satisfied. As a result, the required types for x (U_1, U_4) are replaced with its actual type. The type checker removes the two requirements on x and propagates only the remaining requirements on f .

Finally, the type checker encounters the λ -abstraction on f and operates as in the case of x . That is, the required types of f are replaced with its actual type ($\alpha \rightarrow Num$) and the two remaining requirements on f are removed from the requirement set R , which is now empty. The co-contextual type checker deduces the same type as the context-based type checker and all context requirements are satisfied. Any remaining context requirements in the root node of a program would indicate a type error.

3. Co-contextual PCF

In this chapter, we describe the construction of a co-contextual type system for PCF. To obtain a co-contextual type system from a contextual counterpart the basic technique of dualism is introduced as described in [EBK⁺15]. The technique is demonstrated in the simple setting of PCF. We will further extend the technique along with more complex language features in the following chapters.

Initially, this chapter provides the formal basis of the co-contextual type systems, in terms of syntax and semantics. Also, operations on the requirements are introduced, which are dual to the operations on the typing context. Then, we provide the translation of the PCF typing rules to co-contextual ones and provide their respective equivalence proofs. We obtain an incremental type checker for PCF from the co-contextual one. We apply the techniques of memoization, incremental constraint solving, and eager substitution to the co-contextual type checker. The type checker was implemented in Scala, which we briefly describe next. Finally, we compare the performance of the incremental and non-incremental co-contextual PCF type checkers against the performance of the contextual one.

3.1. Constructing Co-Contextual Type Systems

We can systematically construct co-contextual type rules from context-based type rules. The core idea is to eliminate the context and its propagation and instead to introduce context requirements that are propagated upward the typing derivation. The expressions and types involved in a type rule do not change. In this section, we illustrate how to derive a co-contextual type system for PCF.

Let us consider the type rule for variables first. Traditionally, the PCF rule for variables looks like this:

$$\text{T-VAR} \frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

To co-contextualize this rule, we have to define it without a context. As a consequence, we cannot determine the type T of variable x , which depends on the binding and the usage context of x . To resolve this situation and to define a co-contextual type rule for variables, we apply a trick known from type inference: We generate a fresh type variable and use it as placeholder for the actual type of x . When more information about the type of x becomes available later on during type checking, we use type constraints and unification to retroactively refine the type of variable references. For instance, in the example from Section 2.2, we use fresh type variables U_0 and U_1 for the variable references f and x . Later, when checking the application of f to x , we refine the type of f by adding the

3. Co-contextual PCF

type constraint $U_0 = U_1 \rightarrow U_2$, where U_2 is a fresh type variable itself. As this shows, a co-contextual type checker discovers type refinements in the form of constraints during type checking. We use the metavariable U for type variables that are unification variables as stated in the introduction (Part I) and we use the metavariable X for user-defined type variables when they occur later on:

$T ::= \dots \mid U$ unification variables

As shown in the previous chapter, we assume that the original type rules are given in a constraint-based style and take this as starting point for deriving co-contextual type rules. Co-contextual type rules use judgments of the form $e : T \mid C \mid R$, where e is the expression under analysis and T is the type of e if all type constraints in set C hold and all requirements in set R are satisfied. We can systematically construct co-contextual type rules for PCF from constraint-based contextual type rules using dualism. That is, we replace downward-propagated contexts with upward-propagated context requirements and we replace operations on contexts by their dual operations on context requirements as described in the following subsection.

3.1.1. Co-Contextual Syntax and Operations

We propose to use duality as a generic method for deriving co-contextual type systems. Figure 3.1 summarizes contextual and co-contextual syntax and operations for PCF. The syntax of context requirements is analogous to the syntax of typing contexts. We represent context requirements as a set of type bindings $x : T$. Importantly, context requirements are not ordered and we maintain the invariant that there is at most one binding for x in any context-requirements set.

Contextual	Co-contextual
Judgment $\Gamma \vdash e : T \mid C$	Judgment $e : T \mid C \mid R$
Context syntax $\Gamma ::= \emptyset \mid \Gamma; x : T$	Requirements $R \subset x \times T$ map variables to their types
Context lookup $\Gamma(x) = T$	Requirement introduction $R = \{x : U\}$ with fresh unification variable U
Context extension $\Gamma; x : T$	Requirement satisfaction $R - x$ if $(R(x) = T)$ holds
Context duplication $\Gamma \rightarrow (\Gamma, \Gamma)$	Requirement merging $merge(R_1, R_2) = R _C$ if all constraints $(T_1 = T_2) \in C$ hold
Context is empty $\Gamma = \emptyset$	No unsatisfied requirements $R \stackrel{!}{=} \emptyset$

Figure 3.1.: Operations on contexts and their co-contextual correspondence.

The first contextual operation is context lookup, which we translate into the dual operation of introducing a new context requirement. The context requirement declares that variable x must be well-typed and has type U , which is a fresh unification variable. Note the difference between co-contextual type checking and traditional type inference: Type inference generates a single fresh unification variable U when variable x is introduced (for example, by a λ -abstraction) and coordinates the typing of x via the context. In

3.1. Constructing Co-Contextual Type Systems

contrast, a co-contextual type system generates a new fresh unification variable for every reference of x in the syntax tree. Consequently, co-contextual type checkers typically produce more unification variables than context-based type inference, but they require no coordination.

The second operation is the extension of a context Γ with a new binding $x:T$. The co-contextual operation must perform the dual operation on context requirements, that is, eliminate a context requirement and reduce the set of context requirements. When eliminating a context requirement, it is important to validate that the requirement actually is satisfied. To this end, a co-contextual type system must check that the type of x that is required by the context requirements $R(x)$ is equivalent to T , the type that the original type rule assigned to x . If the constraint solver later finds that $R(x) = T$ does not hold, the type system has identified a context requirement that does not match the actual context. This indicates a type error.

The third operation is the duplication of a typing context, typically to provide it as input to multiple premises of a type rule. Context duplication effectively coordinates typing in the premises. The dual, co-contextual operation merges the context requirements of the premises, thus computing a single set of context requirements to propagate upward. Since the context requirements of the premises are generated independently, they may disagree on requirements for variables that occur in multiple subderivations. Accordingly, it is necessary to retroactively assure that the variables get assigned the same type. To this end, we use an auxiliary function $\text{merge}_R(R_1, R_2) = R|_C$ that identifies overlapping requirements in R_1 and R_2 and generates a merged set of requirements R and a set of type-equality constraints C :

$$\begin{aligned} \text{merge}_R(R_1, R_2) &= R|_C \\ \text{where } R &= R_1 \cup \{x : R_2(x) \mid x \in \text{dom}(R_2) \setminus \text{dom}(R_1)\} \\ C &= \{R_1(x) = R_2(x) \mid x \in \text{dom}(R_1) \cap \text{dom}(R_2)\} \end{aligned}$$

Function merge_R is defined such that the merged requirements R favor R_1 in case of an overlap. This choice is not essential since it gets an equality constraint $R_1(x) = R_2(x)$ for each overlapping x anyways. Based on , we assume the existence of an n -ary function also called merge_R that takes n requirement sets as input and merges all of them, yielding a single requirements set and a set of constraints. For more advanced type systems, we will need to refine function merge_R (see Chapter 4).

The final operation to consider is the selection of an empty context. An empty context means that no variables are bound. For example, this occurs when type checking starts on the root expression. Dually, we stipulate that no context requirements may be left unsatisfied. Note that while the contextual operation selects the empty context, the co-contextual counterpart asserts that subderivations yield an empty requirement set. The difference is that a contextual type check fails when an unbound reference is encountered, whereas the co-contextual type check fails when the context requirement of an unbound variable cannot be satisfied.

We defined the translation from contextual operations to co-contextual operations in a compositional manner by applying duality. As a result, our translation is applicable to compound contextual operations, such as duplicating an extended context or extending

3. Co-contextual PCF

$$\begin{array}{c}
\text{T-Num} \frac{}{n : \text{Num} \mid \emptyset \mid \emptyset} \quad \text{T-Var} \frac{U \text{ is fresh}}{x : U \mid \emptyset \mid x : U} \\
\\
\text{T-Abs} \frac{e : T_2 \mid C \mid R \quad C_x = \{T_1 = R(x) \mid \text{if } x \in \text{dom}(R)\}}{\lambda x : T_1. e : T_1 \rightarrow T_2 \mid C \cup C_x \mid R - x} \\
\\
\text{T-App} \frac{e_1 : T_1 \mid C_1 \mid R_1 \quad e_2 : T_2 \mid C_2 \mid R_2 \quad U \text{ is fresh} \quad \text{merge}_R(R_1, R_2) = R|_C}{e_1 e_2 : U \mid C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow U\} \cup C \mid R} \\
\\
\text{T-Fix} \frac{e : T \mid C \mid R \quad U \text{ is fresh}}{\text{fix } e : U \mid C \cup \{T = U \rightarrow U\} \mid R} \\
\\
\text{T-If0} \frac{e_1 : T_1 \mid C_1 \mid R_1 \quad e_2 : T_2 \mid C_2 \mid R_2 \quad e_3 : T_3 \mid C_3 \mid R_3 \quad \text{merge}_R(R_1, R_2, R_3) = R|_C}{\text{if0 } e_1 e_2 e_3 : T_2 \mid C_1 \cup C_2 \cup C_3 \cup \{T_1 = \text{Num}, T_2 = T_3\} \cup C \mid R}
\end{array}$$

Figure 3.2.: A co-contextual constraint-based formulation of the type system of PCF.

an empty context to describe a non-empty initial context.

3.1.2. Co-Contextual PCF

Figure 3.2 shows the co-contextual type rules of PCF, which we derived according to the method described above. The derivation of T-Num and T-Fix is straightforward as neither rule involves any context operation. In rule T-Var, dual to context lookup, we require x is bound to type U in the context. In rule T-Abs, dual to context extension with x , we check if variable x has been required by the function body e . If there is a requirement for x , we add the constraint $T_1 = R(x)$ and remove that requirement $R - x$. Otherwise C_x is empty and we only propagate the constraints of the body.

The remaining rules T-Add, T-App, and T-If0 use requirement merging dual to the duplication of contexts in the original rules. Each merge gives rise to an additional set of constraints that we propagate. Note that in T-If0 we have to merge requirements from all three subexpressions.

We can use the co-contextual type rules to compute the type of PCF expressions. Given an expression e and a derivation $e : T_e \mid C \mid R$, e is well-typed if R is empty and the constraint C is solvable. If e is well-typed, let $\sigma : U \rightarrow T$ be a type substitution that solves the constraint C . Then, the type of e is $\sigma(T_e)$.

In Chapter 2, we showed an example derivation of co-contextual type checking the expres-

sion $\lambda f : \alpha \rightarrow \text{Num}. \lambda x : \alpha. f\ x + f\ x$. For presentation's sake, we eagerly solved constraints and applied the resulting substitutions. Actually, the co-contextual PCF type system defined here generates the constraint set $\{U_0 = U_1 \rightarrow U_2, U_3 = U_4 \rightarrow U_5, U_2 = \text{Num}, U_5 = \text{Num}, U_0 = U_3, U_1 = U_4\}$ and derives the result type $T = (U_1 \rightarrow \text{Num}) \rightarrow U_1 \rightarrow \text{Num}$. Subsequent constraint solving yields the substitution $\sigma = \{U_0 \mapsto (\alpha \rightarrow \text{Num}), U_1 \mapsto \alpha, U_2 \mapsto \text{Num}, U_3 \mapsto (\alpha \rightarrow \text{Num}), U_4 \mapsto \alpha, U_5 \mapsto \text{Num}\}$ and the final result type $\sigma(T) = (\alpha \rightarrow \text{Num}) \rightarrow \alpha \rightarrow \text{Num}$.

We prove that our co-contextual formulation of PCF is equivalent to PCF. To relate derivations on open expressions containing free variables, we demand the context requirements to be a subset of the provided typing context. In particular, we get equivalence for closed expressions by setting $\Gamma = \emptyset$ and $R = \emptyset$. In our formulation, we call a syntactic entity ground if it does not contain unification variables and we write $\Gamma \supseteq R$ if $\Gamma(x) = R(x)$ for all $x \in \text{dom}(R)$. The following theorem establishes the equivalence of the two formulations of PCF:

Theorem 1. *A program e is typeable in contextual PCF if and only if it is typeable in co-contextual PCF:*

$\Gamma \vdash e : T \mid C$ and $\text{solve}(C) = \sigma$ such that

$\sigma(T)$ and $\sigma(\Gamma)$ are ground

if and only if

$e : T' \mid C' \mid R$ and $\text{solve}(C') = \sigma'$ such that

$\sigma'(T')$ and $\sigma'(R)$ are ground

If e is typeable in contextual and co-contextual PCF as above, then $\sigma(T) = \sigma'(T')$ and $\sigma(\Gamma) \supseteq \sigma'(R)$.

Proof. By structural induction on e . A detailed proof is shown in Appendix A. □

3.2. Incremental Type Checking

Incremental computations often follow a simple strategy: Whenever an input value changes, transitively recompute all values that depend on a changed value until a fixed-point is reached [MEK⁺14]. We apply the same strategy to incrementalize type checking. As we will see, the avoidance of contextual coordination in co-contextual type systems makes dependency tracking particularly simple and thus enables incremental type checking.

For illustration, we use the following PCF expressions as a running example throughout this section. To simplify the example, we added subtraction to PCF in a straightforward manner.

$\text{mul} = \text{fix } (\lambda f : \text{Num} \rightarrow \text{Num} \rightarrow \text{Num}.$
 $\quad \lambda m : \text{Num}. \lambda n : \text{Num}. \text{if0 } m\ 0\ (n + f\ (m - 1)\ n))$
 $\text{notfac} = \text{fix } (\lambda f : \text{Num} \rightarrow \text{Num}.$
 $\quad \lambda n : \text{Num}. \text{if0 } (n - 1)\ 1\ (\text{mul } n\ (f\ (n - 2))))$

The first expression mul defines multiplication on top of addition by recursion on the first argument. The second expression notfac looks similar to the factorial function, but it is undefined for $n = 0$ and the recursive call subtracts 2 instead of 1 from n . Below,

3. Co-contextual PCF

we exemplify how to incrementally type check these expression when changing *notfac* to match the factorial function. The initial type check of *mul* and *notfac* generate 11 and 12 constraints, respectively.

3.2.1. Basic Incrementalization Scheme

In the previous sections, we defined co-contextual type rules using a typing judgment of the form $e : T \mid C \mid R$. In this section, we additionally assume that the type rules are given in an algorithmic form, that is, they are syntax-directed and their output is fully determined by their input. Therefore, another way to understand the type rules is as a function $check_1 : e \rightarrow T \times C \times R$ that maps an expression to its type, typing constraints, and context requirements.

Note that $check_1$ only depends on the expression and is independent of any typing context. This means that an expression e is assigned to the same type, constraints, and requirements independent of the usage context (modulo renaming of unification variables). For example, $check_1(e)$ yields the same result no matter if it occurs as a subderivation of $e + 5$ or $\lambda x : Num. e$. The only difference is that some usage scenarios may fail to satisfy the constraints or context requirements generated for e . Accordingly, when an expression changes, it is sufficient to propagate these changes up the expression tree to the root node; siblings are never affected.

Based on this observation, we can define a simple incrementalization scheme for co-contextual type checking:

E = set of non-memoized subexpressions in *root*
 E^C = transitive closure of E under parent relationship
 E_p^C = sequentialize E^C in syntax-tree post-order
 for each $e \in E_p^C$
 recompute and memoize the type of e
 using the memoized types of its subexpressions

That is, we visit all subexpressions that have changed and all subexpressions that (transitively) depend on changed subexpressions. We use a post-order traversal (i.e., bottom-up) to ensure we visit all changed subexpressions of an expression before recomputing the expression's type. Accordingly, when we type check an expression in E_p^C , the types (and constraints and requirements) of all subexpressions are already available through memoization. We present an efficient implementation of this scheme in Section 3.3.

To illustrate, let us consider the example expression *notfac* from above. First, we observe that a change to *notfac* never affects the typing of *mul*, which we can fully reuse. When we change the *if*-condition of *notfac* from $n - 1$ to n , our incremental type checker recomputes types for the expressions $E_p^C = \langle n_{cond}, if0, \lambda n, \lambda f, fix \rangle$, starting at the changed condition. Importantly, we reuse the memoized types from the initial type check of the *else*-branch. When instead changing the subtraction in the recursive call of *notfac* from $n - 2$ to $n - 1$, the type checker recomputes types for the expressions $E_p^C = \langle 1, n - 1, app_f, app_{mul2}, if0, \lambda n, \lambda f, fix \rangle$. While this change entails lots of recomputation,

we still can reuse results from previous type checks for the application of *mul* to its first argument and for the *if*-condition. Finally, consider what happens when we introduce a type error by changing the binding λn to λx . A contextual type checker would have to reconstruct the derivation of the body because the context changed and n now is unbound. In contrast, our incremental type checker can reuse all constraints from the body of the abstraction and only has to regenerate the constraint for *fix*. The table below summarizes the number of regenerated and reused constraints. Our performance evaluation in Section 3.4 confirms that incrementalization of type checking can improve type-checking performance significantly.

Changes in <i>notfac</i>	Regenerated constraints	Reused constraints	Reused from <i>mul</i>
<i>if</i> -cond. $n - 1 \mapsto n$	4	6	11
rec. call $n - 2 \mapsto n - 1$	9	3	11
binding $\lambda n \mapsto \lambda x$	1	11	11

3.2.2. Incremental Constraint Solving

In the previous subsection, we have developed an incremental type checker $check_1 : e \rightarrow T \times C \times R$. However, all that $check_1$ actually does is to generate constraints and to collect context requirements. That is, even when run incrementally, $check_1$ yields a potentially very large set of constraints that we have to solve in order to compute the type of e . We only incrementalized constraint generation but not constraint solving so far.

Designing an incremental constraint solver is difficult because unification and substitution complicate precise dependency tracking and updating of previously computed solutions when a constraint is removed. However, since type rules only add constraints but never skip any constraints from subderivations, we can solve the intermediate constraint systems and propagate and compose their solutions. To this end, we assume the existence of two functions:

$$\begin{aligned} solve & : C && \rightarrow \sigma \times C^- \times C^? \\ finalize & : \sigma \times C^- \times C^? && \rightarrow \sigma \times C^- \end{aligned}$$

Function *solve* takes a set of constraints and produces a partial solution $\sigma : U \rightarrow T$ mapping unification variables to types, a set of unsatisfiable constraints C^- indicating type errors, and a set of possibly satisfiable constraints $C^?$ that may or may not hold in a larger context. The intermediate solutions are not necessarily ground substitutions since domain U is a subset of domain T . While intermediate solutions discharge equality constraints, they do not necessarily eliminate all unification variables. Function *finalize* assumes a closed world and either solves the constraints in $C^?$ or marks them as unsatisfiable. We include $C^?$ for type systems that require (partially) context-sensitive constraint solving. For example, for PCF with subtyping, $C^?$ contains constraints that encode lower and upper bounds on type variables. In practice, to achieve good performance, it is important

3. Co-contextual PCF

to normalize the constraints in $C^?$ and represent them compactly.

Using *solve*, we can modify type rules such that they immediately solve the constraints they generate. Type rules then propagate the composed solutions from their own constraints and all subderivations. For example, we obtain the following definition of T-APP, where we write $\sigma_1 \circ \sigma_2$ to denote substitution composition.¹

$$\begin{array}{c}
 e_1 : T_1 \mid \sigma_1 \mid C_1^- \mid C_1^? \mid R_1 \\
 e_2 : T_2 \mid \sigma_2 \mid C_2^- \mid C_2^? \mid R_2 \\
 U \text{ is fresh} \quad \text{merge}_R(R_1, R_2) = R|_C \\
 \text{solve}(C \cup \{T_1 = T_2 \rightarrow U\}) = (\sigma_3, C_3^-, C_3^?) \\
 \sigma_1 \circ \sigma_2 \circ \sigma_3 = \sigma \\
 \text{T-APP} \frac{}{e_1 e_2 : U \mid \sigma \mid C_1^- \cup C_2^- \cup C_3^- \mid C_1^? \cup C_2^? \cup C_3^? \mid R}
 \end{array}$$

This gives rise to an incremental type checker $check_2 : e \rightarrow T \times \sigma \times C^- \times C^? \times R$ using the incrementalization scheme from above. In contrast to $check_1$, $check_2$ actually conducts incremental constraint solving since we incorporated constraint solving into the type rules. The only constraints we have to solve non-incrementally are those in $C^?$, for which we use *finalize* on the root node.

Let us revisit the example expression *notfac* from above. When changing the *if*-condition or recursive call, we obtain the same number of regenerated constraints. However, instead of reusing previously generated constraints, $check_2$ reuses previously computed solutions. This means that after a change, we only have to solve the newly generated constraints. For example, after the initial type check, we never have to solve the constraints of *mul* again, because it does not change. As our performance evaluation in Section 3.4 shows, incremental constraint solving significantly improves incremental performance and, somewhat surprisingly, also the performance of the initial type check.

3.2.3. Eager Substitution

Co-contextual type rules satisfy an interesting property that enables eager substitution of constraint solutions. Namely, the constraints in independent subexpressions yield non-conflicting substitutions.

This statement holds for two reasons. First, every time a type rule requires a fresh unification variable U , this variable cannot be generated fresh by any other type rule. Thus, U cannot occur in constraints generated while type checking any independent subexpression. Hence, there can be at most one type assigned to U by a constraint. Second, we formulated co-contextual type rules in a way that strictly separates user-defined type variables X from unification variables U generated by type rules. While

¹Similar to requirements merging, the composition of substitutions can yield additional constraints when the domains of the substitutions overlap, which can be easily resolved by additional constraint solving. We omit this detail to keep the presentation concise.

a unification variable is generated fresh once, a user-defined type variable X can occur multiple times in the syntax tree, for example within type annotations of different λ -abstractions. Thus, type checking independent subexpressions can yield constraints that jointly constrain X . When we solve such constraints independently, there is the danger of assigning different types to X , which would require coordination. However, since the substitutions computed by *solve* map unification variables to types and user-defined type variables are not considered unification variables, this situation cannot occur.

$check_2$ propagated substitutions up the tree. However, as substitutions are non-conflicting, we can eagerly apply the substitution within the type rule, thus avoiding its propagation altogether. For example, we can redefine type rule T-APP as follows:

$$\text{T-APP} \frac{\begin{array}{l} e_1 : T_1 \mid C_1^- \mid C_1^? \mid R_1 \quad e_2 : T_2 \mid C_2^- \mid C_2^? \mid R_2 \\ U \text{ is fresh} \quad \text{merge}_R(R_1, R_2) = R|_C \\ \text{solve}(C \cup \{T_1 = T_2 \rightarrow U\}) = (\sigma, C_3^-, C_3^?) \\ C^- = C_1^- \cup C_2^- \cup C_3^- \quad C^? = C_1^? \cup C_2^? \cup C_3^? \end{array}}{e_1 e_2 : \sigma(U) \mid \sigma(C^-) \mid \sigma(C^?) \mid \sigma(R)}$$

In this type rule, the substitution σ is not propagated. Instead, we directly apply it to all components of the type rule's result. By applying the substitution, we eliminate all unification variables $U \in \text{dom}(\sigma)$ from the result. But then, as unification variables are not shared between independent trees, there is no need to propagate the substitution itself.

This design yields an incremental type-check function $check_3 : e \rightarrow T \times C^- \times C^? \times R$. Compared to the previous version, this type checker can improve both the initial and incremental performance because it avoids the propagation, composition, and memoization of substitutions. As our performance evaluation in Section 3.4 shows, we achieve best performance by using a hybrid approach that stores and propagates small substitutions but eagerly applies larger substitutions that bind ten or more variables.

3.3. Technical Realization

We have developed efficient incremental type checkers in Scala for PCF. The source code is available online at

<https://github.com/seba--/incremental>.

In-tree memoization. Implementing incremental type checking as described in the previous section requires memoization to map expressions to previously computed typing information. When type checking an expression, we first check if a type has been previously memoized, in which case we return this type without further computation. For better memory and time performance, we do not use a lookup table, but memoize previously computed types directly in the syntax tree. As a consequence of in-tree memoization,

3. Co-contextual PCF

trees that are equivalent but represented as separate objects do not share their memoized types. To avoid this, our representation supports syntax representations with sharing based on acyclic object graphs. To support tree changes efficiently, our expressions store a mutable list of child expressions. We encode an incremental change to a syntax tree via an update to the parent’s list of child expressions. If the new child has a memoized type, we keep it and invalidate the parent’s type. If the child has no memoized type, dependency tracking will invalidate the parent’s type automatically.

Efficient incremental type checking. The incrementalization scheme described in Section 3.2.1 (1) selects all non-memoized subexpressions, (2) closes them under parent relationship, (3) orders them in syntax-tree post-order, and (4) iterates over them to recompute and memoize types. To implement this scheme efficiently, we merge these operations into a single post-order tree traversal. During the traversal, we recompute and memoize the type of a subexpression if the type was not computed before or if the type of any direct subexpression was recomputed during the same traversal. This traversal has the same semantics as the scheme presented before, but avoids the materialization and iteration over intermediate collections of subexpressions.

Constraint systems. The implementation details of the constraint systems heavily depend on the supported language constructs. To simplify the definition of incremental type checkers, our framework provides an abstract constraint-system component that captures the commonalities of different constraint systems, but abstracts from the internal representation and processing of constraints.

For PCF, the constraint system is straightforward and simply solves equational constraints by unification.

3.4. Performance Evaluation

We evaluate the performance of non-incremental and incremental co-contextual type checking through micro-benchmarking. Specifically, we compare the performance of the following five PCF type checkers on a number of synthesized input programs:²

- DU: Standard contextual constraint-based down-up type checker (base line) that propagates contexts downward and types upward.
- BU1: Co-contextual bottom-up type checker with incremental constraint generation (Section 3.2.1).
- BU2: Like BU1 but with incremental constraint solving (Section 3.2.2).
- BU3: Like BU2 but with eager substitution (Section 3.2.3).
- BU4: Like BU3 but only eagerly substitute when $|\sigma| \geq 10$.

²The benchmarking code and raw data is available online at <https://github.com/seba--/incremental>.

We expect to show two results with our evaluation. First, when running non-incrementally, our co-contextual type checkers have performance comparable to the standard contextual type checker DU. Second, after an incremental program change, our co-contextual type checkers outperform the standard contextual type checker DU. Moreover, our evaluation provides some initial data for comparing the performance of the co-contextual type checkers BU1 to BU4.

Input data. We run the type checkers on synthesized input expressions of varying sizes. We use balanced binary syntax trees $T(N^h, l_1 \dots l_n)$ of height h with binary inner nodes N and leaves $l_1 \dots l_n$, where $n = 2^{h-1}$. In particular, we run the type checkers for heights $h \in \{2, 4, 6, 8, 10, 12, 14, 16\}$, inner nodes $N \in \{+, \text{app}\}$, and leaf sequences consisting of numeric literals $(1 \dots n)$, a single variable $(x \dots x)$, or a sequence of distinct variables $(x_1 \dots x_n)$.

We chose these trees as input to explore the impact of context requirements. Trees with numeric literals $(1 \dots n)$ as leaves are type checked under an empty typing context and yield an empty requirement set. In contrast, trees with a single variable $(x \dots x)$ as leaves are type checked under a typing context with a single entry, but co-contextual type checking introduces a distinct unification variable for each variable occurrence and has to perform lots of requirement merging yielding additional constraints. Finally, trees with a sequence of distinct variables $(x_1 \dots x_n)$ are type checked under a typing context with n entries and also yield a requirement set with n entries. In the latter case, requirement merging does not yield any additional constraints because all variables are distinct. We chose addition and application as inner nodes because they yield constraints of different complexity $\{T_1 = \text{Num}, T_2 = \text{Num}\}$ and $\{T_1 = T_2 \rightarrow U\}$, respectively.

We do not use sharing of subtrees, thus our largest trees have $2^{16} - 1 = 65535$ nodes. For comparison, the largest source file of Apache Ant 1.9.4 has 17814 Java syntax-tree nodes. In our synthesized expressions, all variables occurring in a λ -expression at the top of the generated tree are bound. Instead of type annotations, we rely on the type checkers to infer the type of bound variables. Some of the synthesized expressions are ill-typed, namely when applying a number in place of a function and when applying a variable to itself. This allows us to also evaluate the run time of failing type checks. We leave the evaluation of co-contextual type checkers on real-world programs written in modern languages like Java for future work.

Experimental setup. We measure the performance of a type checker in terms of the number of syntax-tree nodes it processes per millisecond. We use ScalaMeter³ to measure the wall-clock run times of our Scala implementations of the above type checkers. ScalaMeter ensures proper JVM warm up, takes a sequence of measurements, eliminates outliers, and computes the mean run time of the rest. We performed the benchmark on a 3Ghz octa-core Intel Xeon E5-1680 machine with 12GB RAM, running Mac OS X 10.10.4, Scala 2.11.5 and Oracle JDK 8u5 with a fixed JVM heap size of 4GB.

Based on the mean run time and the size of the input expression tree, we calculate the nonincremental performance $\frac{\#nodes=2^h-1}{run\ time\ in\ ms}$ of a type checker on different inner-node

³<http://scalameter.github.io>

3. Co-contextual PCF

and leaf-node combinations. For each combination, we report the mean performance over all height configurations as well as the speedup relative to DU. Moreover, we report the overall performance of each checker as the mean performance over all tree shapes. For BU1, we were not able to measure the performance for $h = 14$ and $h = 16$ due to high garbage-collection overheads and consequent timeouts.

For measuring incremental performance, we fix the height of the input syntax tree at $h = 16$ (due to timeouts, we excluded BU1 from this experiment). We first perform a full initial type check. To simulate incremental changes, we invalidate all types stored for the left-most subtree of height $k \in \{2, 4, 6, 8, 10, 12, 14, 16\}$. We measure the wall-clock run times for rechecking the partially invalidated syntax trees. We calculate the mean performance of a recheck $\frac{\#nodes=2^h-1}{run\ time\ in\ ms}$ relative to the total size of the syntax tree. We report the mean performance over all height configurations k , the speedup relative to DU, and the overall performance as the mean performance over all tree shapes.

Tree	DU	BU1	BU2	BU3	BU4
$T(+^h, 1 \dots n)$	1078.37	836.70 (0.78)	1164.15 (1.08)	736.81 (0.68)	1109.52 (1.03)
$T(+^h, x \dots x)$	714.74	91.37 (0.13)	267.46 (0.37)	254.92 (0.36)	241.36 (0.34)
$T(+^h, x_1 \dots x_n)$	188.27	67.77 (0.36)	218.55 (1.16)	176.66 (0.94)	211.82 (1.13)
$T(\mathbf{app}^h, 1 \dots n)$	219.27	94.56 (0.43)	302.76 (1.38)	357.14 (1.63)	294.18 (1.34)
$T(\mathbf{app}^h, x \dots x)$	185.84	27.17 (0.15)	68.38 (0.37)	127.96 (0.69)	104.09 (0.56)
$T(\mathbf{app}^h, x_1 \dots x_n)$	119.41	58.33 (0.49)	130.91 (1.10)	132.23 (1.11)	153.57 (1.29)
overall performance	417.65	195.98 (0.39)	358.70 (0.91)	297.62 (0.90)	352.42 (0.95)

(a) Nonincremental performance in nodes per millisecond (speedup relative to DU).

Tree	DU	BU1	BU2	BU3	BU4
$T(+^h, 1 \dots n)$	1507.64	n/a	37028.61 (24.56)	28532.45 (18.93)	36277.34 (24.06)
$T(+^h, x \dots x)$	1147.44	n/a	2852.65 (2.49)	9699.62 (8.45)	11512.60 (10.03)
$T(+^h, x_1 \dots x_n)$	386.18	n/a	1165.87 (3.02)	1168.82 (3.03)	1670.72 (4.33)
$T(\mathbf{app}^h, 1 \dots n)$	224.08	n/a	1564.35 (6.98)	1911.00 (8.53)	2194.19 (9.79)
$T(\mathbf{app}^h, x \dots x)$	223.05	n/a	78.55 (0.35)	795.25 (3.57)	777.94 (3.49)
$T(\mathbf{app}^h, x_1 \dots x_n)$	124.49	n/a	609.23 (4.89)	728.50 (5.85)	1178.55 (9.47)
overall performance	602.15	n/a	7216.54 (7.05)	7139.27 (8.06)	8935.22 (10.19)

(b) Incremental performance in nodes per millisecond (speedup relative to DU).

Figure 3.3.: Nonincremental and incremental type-checking performance for PCF.

Nonincremental performance. Figure 3.3(a) shows the nonincremental performance numbers for different tree configurations with speedups relative to DU in parentheses. First, note that all type checkers except BU1 are relatively fast in that they process a

syntax tree of height 16 with 65535 nodes in between 157ms and 220ms (number of nodes divided by nodes per ms). BU1 is substantially slower, especially considering that we had to abort the executions for $h = 14$ and $h = 16$.

On average, type checker DU processes 417.65 syntax-tree nodes per millisecond. BU2, BU3, and BU4 perform only slightly worse than DU. By inspecting individual rows in Figure 3.3(a), we see that co-contextual type checkers actually outperform DU in many cases. For example, BU4 is faster than DU when leaves are numeric expressions or distinct variables. However, all co-contextual type checkers perform comparatively bad when all leaves of the syntax refer to a single variable because a large number of requirement merging is needed. For example, in a tree of height 16 we have $2^{15} = 32768$ references to the same variable and $2^{16} - 2^{15} - 1 = 32767$ calls to *merge*, each of which generates a constraint to unify the types of the variable references. In summary, we can say that BU2 and BU4 have run-time costs similar to those of DU, but their performance varies with respect to variable occurrences.

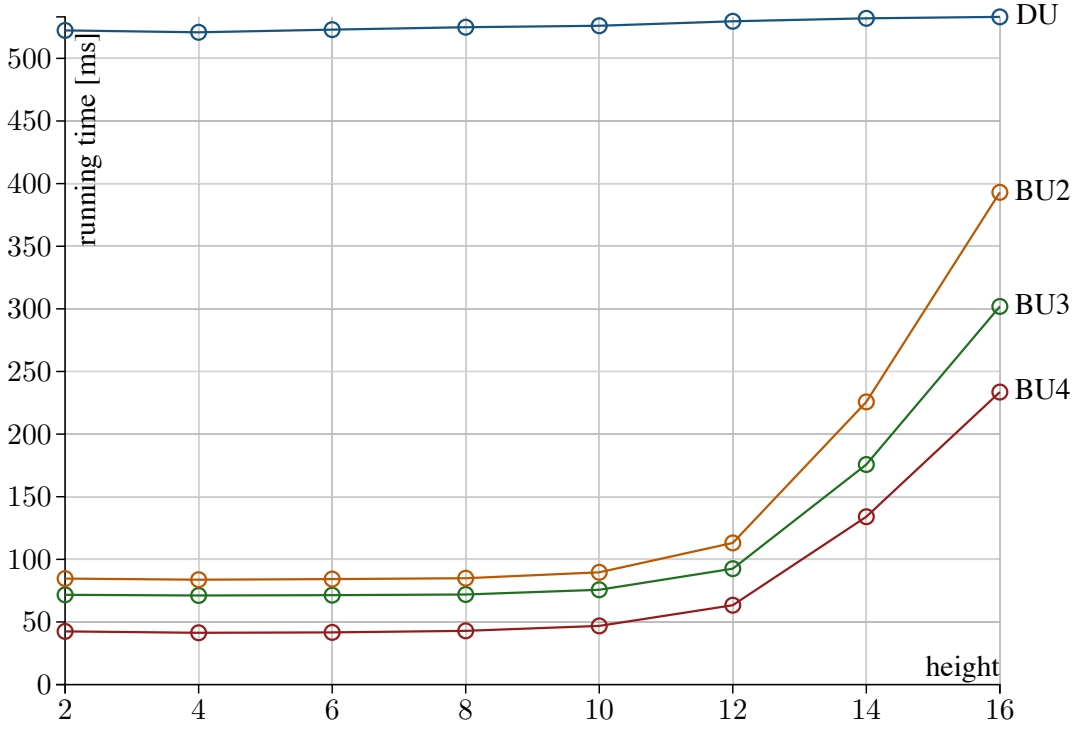


Figure 3.4.: Incremental running time on $T(\text{app}^{16}, x_1 \dots x_n)$ for changes of size $2^{\text{height}} - 1$.

Incremental performance. Figure 3.4 shows the incremental performance of DU (blue), BU2 (orange), BU3 (green), and BU4 (red) on $T(\text{app}^{16}, x_1 \dots x_n)$. The x-axis shows the height of the invalidated subexpression; the y-axis shows the run time of the four type checkers. DU does not support incremental type checking and therefore takes the same time to recheck the input of size $2^{16} - 1$ independent of the size of the invalidated

3. Co-contextual PCF

subexpression. In contrast, BU2, BU3, and BU4 run considerably faster. Especially for small changes, incremental type checking provides large speedups. However, the graph also reveals that the incremental type checking does not scale down to small changes linearly. Instead, we observe that incremental rechecking takes roughly the same time when invalidating a subexpression of height $k \in \{2, 4, 6, 8\}$. This is because our incremental type checkers need to traverse the whole syntax tree once in order to find the invalidated subexpression. Thus, even a small change incurs the cost of traversing the syntax tree once.

Figure 3.3(b) presents a bigger picture of the incremental performance, where we report the mean performance over all height configurations k of the invalidated subexpression. Since DU does not support incremental type checking, it has to recheck the whole syntax tree nonincrementally for every change. The numbers for DU differ from the numbers reported in Figure 3.3(a) because we fixed the height to $h = 16$. For the co-contextual type checkers, we see a significant performance improvement of up to 24.56x. Incremental type checkers BU3 and BU4 also achieve good speedups when all leaves refer to the same variable, which yielded slowdowns in the nonincremental case.

The co-contextual type checkers BU2, BU3, BU4 follow different substitution strategies, where the results indicate that the choice of strategy noticeably influences performance. Deferring substitutions until the final step (BU2) and immediate substitutions (BU3) compare worse to a balanced strategy (BU4), where the type checker defers substitution until incremental constraint solving generates a certain amount of solutions.

4. Simple Extensions of PCF

In this chapter, we present co-contextual type systems for the following extensions of PCF: records, parametric polymorphism, and structural subtyping. We show how to systematically adapt the co-contextual type system of PCF in order to support each extension. We illustrate the addition of types, expressions and constraints correspondingly for each extension. Initially, we present the additions required to support records. Next, we consider parametric polymorphism, which is more complex and requires the introduction of universal types. Finally, we consider structural subtyping, which is equally complex and requires the adaptation of the merge operation.

4.1. Records

Many type-system features do not change or inspect the typing context. We can define such features as simple extensions of the co-contextual PCF type system. As a representative for such features, we present an extension with records, using the following extended syntax for expressions, types, and type constraints:

$$\begin{aligned} e &::= \dots \mid \{l_i = e_i\}_{i \in 1 \dots n} \mid e.l && \text{record expressions} \\ T &::= \dots \mid \{l_i : T_i\}_{i \in 1 \dots n} && \text{record types} \\ c &::= \dots \mid T.l = T && \text{field type constraint} \end{aligned}$$

$$\begin{array}{c} \text{T-RECORD} \frac{e_i : T_i \mid C_i \mid R_i \text{ for } i \in 1 \dots n}{\text{merge}_R(R_1, \dots, R_n) = R|_{C_R} \bigcup_{i \in 1 \dots n} C_i = C} \\ \frac{}{\{l_i = e_i\}_{i \in 1 \dots n} : \{l_i : T_i\}_{i \in 1 \dots n} \mid C \cup C_R \mid R} \\ \\ \text{T-LOOKUP} \frac{e : T \mid C \mid R \quad U \text{ is fresh}}{e.l : U \mid C \cup \{T.l = U\} \mid R} \end{array}$$

Figure 4.1.: Co-contextual type rules for records.

The additional type rules for records with respect to co-contextual PCF are illustrated in Figure 4.1. Type rule T-RECORD defines how to type check record expressions $\{l_i = e_i\}_{i \in 1 \dots n}$. The type of the record expression is a record type, where each label is associated type T_i of subexpression e_i . To type check the subexpressions e_i , a traditional contextual type rule for

4. Simple Extensions of PCF

record expressions uses a replica of its typing context for each subderivation. In accordance with the definitions presented in the previous chapter in Figure 3.1, a co-contextual type rule for record expressions merges the requirements of all subderivations. Type rule $T\text{-RECORD}$ in Figure 4.1 merges the requirements R_i into the fully merged requirements set R with additional constraints C_R . We propagate the merged requirements, the additional constraints, and the original constraints C_i of the subderivations.

Type rule $T\text{-LOOKUP}$ defines type checking for field lookup $e.l$. In our setting, we cannot simply match the type T of e to extract the type of field l because T may be a unification variable that is only resolved later after solving the generated constraints. Instead, we use a constraint $T.l = U$ that expresses that T is a record type that contains a field l of type U .

Similar to records, we can easily define further simple extensions of our co-contextual formulation of PCF, such as variants or references.

4.2. Parametric Polymorphism

In the following, we present the co-contextual formulation of PCF extended with parametric polymorphism. This extension is interesting with respect to our co-contextual formulation because (i) the type checker can encounter type applications without knowledge of the underlying universal type and (ii) parametric polymorphism requires further context operations to ensure that there are no unbound type variables in a program. To support parametric polymorphism, we first add new syntactic forms for type abstraction and application as well as for type variables and universal types.

$$\begin{aligned} e &::= \dots \mid \lambda X. e \mid e[T] \\ T &::= \dots \mid X \mid \forall X. T \mid \forall U. T \\ c &::= \dots \mid T = \{X \mapsto T\} T \mid T = \{U \mapsto T\} T \end{aligned}$$

Note that due to the constraint-based nature of co-contextual type checking, we require support for universal types that quantify over user-supplied type variables X as well as unification type variables U . Importantly, the universal type $\forall U. T$ does not bind the unification variable U ; unification variables are always bound globally. Instead, $\forall U. T$ binds the type variable that will eventually replace U . Moreover, we require new constraints of the form $T_1 = \{X \mapsto T_2\} T_3$ that express that T_1 is the result of substituting T_2 for X in T_3 . We define similar constraints for substituting unification variables. However, their semantics differ in that the constraint solver must delay the substitution of a unification variable until it is resolved to a proper type. For example, the substitution in $T_1 = \{U \mapsto T_2\} X$ must be delayed because it might later turn out that $U = X$. Furthermore, the constraint solver may not substitute user-supplied type variables X as they are not unification variables, hence the constraint $X_1 = X_2 \rightarrow X_2$ does not hold. This distinction of type variables also entails specific rules for the substitution and unification of universal types, which permits the refinement of unification variables even when they appear as binding or bound occurrences.

Since parametric polymorphism introduces bindings for type variables, we also need to track that no unbound type variables occur in a program. A traditional contextual type

checker adds bound type variables to the typing context and checks that all occurrences of type variables are indeed bound. We can use the same strategy as for term variables (Chapter 3) to co-contextualize type-variable handling. In particular, we introduce an additional requirements component for type variables $R^T \subset X$ and extend our typing judgment $e : T \mid C \mid (R, R^T)$ to propagate required type variables R^T . Dual to lookup and introduction of type variables in contextual type checking, we produce type-variable requirements when checking a user-supplied type for well-formedness $\vdash T \text{ ok} \mid R^T$ and we eliminate type-variable requirements when binding a type variable in a type-level λ -abstraction. As before, an expression is only well-typed if all requirements are satisfied, that is, there are neither term-variable nor type-variable requirements on the root of the syntax tree.

$$\begin{array}{c}
\text{T-TABS} \frac{e : T \mid C \mid (R, R^T)}{\lambda X. e : \forall X. T \mid C \mid (R, R^T - X)} \\
\\
\text{T-TAPP} \frac{e : T_1 \mid C \mid (R, R_e^T) \quad U, U_b, U_r \text{ is fresh} \quad \vdash T \text{ ok} \mid R^T}{e[T] : U_r \mid C \cup \{T_1 = \forall U. U_b\} \cup \{U_r = \{U \mapsto T\} U_b\} \mid (R, R_e^T \cup R^T)} \\
\\
\text{T-ABS} \frac{e : T_2 \mid C \mid (R, R_e^T) \quad \vdash T_1 \text{ ok} \mid R_1^T \quad C_x = \{T_1 = R(x) \mid \text{if } x \in \text{dom}(R)\}}{\lambda x : T_1. e : T_1 \rightarrow T_2 \mid C \cup C_x \mid (R - x, R_e^T \cup R_1^T)} \\
\\
\text{T-APP} \frac{e_1 : T_1 \mid C_1 \mid (R_1, R_1^T) \quad e_2 : T_2 \mid C_2 \mid (R_2, R_2^T) \quad U \text{ is fresh} \quad \text{merge}_R(R_1, R_2) = R|_C}{e_1 e_2 : U \mid C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow U\} \cup C \mid (R, R_1^T \cup R_2^T)}
\end{array}$$

Figure 4.2.: Co-contextual type rules for parametric polymorphism.

Figure 4.2 shows the type rules for type abstraction, type application, and term abstraction. Rule T-TABS handles type abstraction $\lambda X. e$. It eliminates type-variable requirements on the bound type variable X and propagates the remaining type-variable requirements $R^T - X$. Rule T-TAPP handles type applications $e[T]$. It checks the subexpression e for well-typedness and the application type T for well-formedness and propagates their combined type-variable requirements $R_e^T \cup R^T$. As the first constraint of T-TAPP stipulates, type application is only well-typed if the type of e is a universal type $\forall U. U_b$. The type of the type application then is the result of substituting T for U in U_b , as the second constraint defines.

Type rule T-ABS in Figure 4.2 is an extended version of the co-contextual PCF rule for λ -abstraction from Chapter 3. Due to the existence of type variables, we added a premise

4. Simple Extensions of PCF

that checks the well-formedness of the type annotation T_1 . We propagate the resulting type-variable requirements together with the type-variable requirements of the function body. Finally, type rule T-APP illustrates how to extend all other rules of PCF such that they merge and propagate type-variable requirements from subexpressions. Note that due to the simplicity of type-variable requirements, the merge operation is simply set union. We would require a more sophisticated merge operation when introducing type variables of different kinds, for example, to realize higher-order polymorphism.

To illustrate these type rules, consider the co-contextual type checking of the polymorphic identity function instantiated for *Num*: $(\lambda X. \lambda x : X. x)[\text{Num}]$.

$$\begin{array}{l}
\text{tapp } [\text{Num}] : U_r \mid \left\{ \begin{array}{l} U_0 = X, \\ \forall X. (X \rightarrow U_0) = \forall U. U_b, \\ U_r = \{U \mapsto \text{Num}\} U_b \end{array} \right\} \mid (\emptyset, \emptyset) \\
| \\
\lambda X : \forall X. (X \rightarrow U_0) \mid \{U_0 = X\} \mid (\emptyset, \emptyset) \\
| \\
\lambda x : X : X \rightarrow U_0 \mid \{U_0 = X\} \mid (\emptyset, \{X\}) \\
| \\
x : U_0 \mid \emptyset \mid (x : U_0, \emptyset)
\end{array}$$

The type checker processes the expression bottom-up. First, it associates a fresh unification variable U_0 to x . Second, the λ -abstraction binds x to type X , which yields the constraint $U_0 = X$ and a type-variable requirement on X . Third, this requirement is immediately discharged by the type-level λ -abstraction that binds X . Finally, the type application rule requires a universal type and computes the result type U_r via a type-substitution constraint. Subsequent constraint solving yields the substitution $\sigma = \{U_0 \mapsto X, U \mapsto X, U_b \mapsto (X \rightarrow X), U_r \mapsto (\text{Num} \rightarrow \text{Num})\}$.

Our type system rejects expressions with unbound type variables. For example, the expression $\lambda f : \text{Num} \rightarrow X. x \ 0$ contains an unbound type variable X . When type checking this expression in our type system, we receive an unsatisfied type-variable requirement that represents this error precisely. Furthermore, despite using constraints, our type system correctly prevents any refinement of universally quantified type variables. For example, our type system correctly rejects the expression $\lambda X. \lambda x : X. x + x$, which tries to refine X to *Num* to perform addition.

4.3. Structural Subtyping

As another extension, we consider a co-contextual formulation of PCF with subtyping. Subtyping is an interesting extension because it affects the semantics of a typing contexts and, hence, context requirements. In particular, subtyping weakens the assumptions about variable bindings $x : T$ in typing contexts. In standard PCF, $(x : T_x; \Gamma)$ means that variable x has *exactly* type T_x . In contrast, in PCF with subtyping, $(x : T_x; \Gamma)$ means that T_x is an *upper bound* of the type of values substitutable for x : All values must at least adhere to T_x . Dually, a type annotation T_x on a λ -abstraction is a lower bound for

the type required for x in subexpressions: Subexpressions can at most require T_x . Thus, subtyping affects the merging and satisfaction of context requirements.

We adapt the definition of merge_R presented in the previous chapter to correctly combine requirements from different subexpressions. Due to subtyping, different subexpressions can require different types on the same variable. In consequence, a variable has to simultaneously satisfy the requirements of all subexpressions that refer to it. That is, when we merge overlapping context requirements, we require the type of shared variables to be a subtype of both originally required types:

$$\begin{aligned} \text{merge}_R(R_1, R_2) &= R|_C \\ \text{where } X &= \text{dom}(R_1) \cap \text{dom}(R_2) \\ U_x &= \text{fresh variable for each } x \in X \\ R &= (R_1 - \text{dom}(R_2)) \cup (R_2 - \text{dom}(R_1)) \\ &\quad \cup \{x : U_x \mid x \in X\} \\ C &= \{U_x <: R_1(x), U_x <: R_2(x) \mid x \in X\} \end{aligned}$$

We do not stipulate a specific subtyping relation. However, we add new forms of type constraints with standard semantics to express subtyping, joins, and meets:

$$\begin{aligned} c ::= & \dots \mid T <: T && \text{subtype constraint} \\ & \mid T = T \vee T && \text{least upper bound (join)} \\ & \mid T = T \wedge T && \text{greatest lower bound (meet)} \end{aligned}$$

A least upper bound constraint $T_1 = T_2 \vee T_3$ states that type T_1 is the least type in the subtype relation such that both T_2 and T_3 are subtypes of T_1 . A greatest lower bound constraint $T_1 = T_2 \wedge T_3$ states that the type T_1 is the greatest type in the subtype relation such that both T_2 and T_3 are supertypes of T_1 .

Figure 4.3 shows the co-contextual rules for PCF enriched with subtyping. Only the rules for λ -abstractions, applications, conditionals, and fixpoints change with respect to the co-contextual PCF type system in Chapter 3. First, consider the rule for λ -abstraction T-ABS . As discussed above, a context requirement on x only describes an upper bound on the declared type of x . Or conversely, the declared type of a variable x is a lower bound on what subexpressions can require for x . Accordingly, we replace the type-equality constraint by a subtype constraint $T_1 <: R(x)$.

The other rules T-APP , T-IF0 , and T-FIX are straightforward extensions to allow for subtyping. In rule T-APP , we allow the argument to be of a subtype of the function's parameter type as usual. Rule T-IF0 declares the type of the conditional to be the least upper bound of the two branch types, which we express by the least upper bound constraint $U = T_1 \vee T_2$. In rule T-FIX , we permit the fixed function to produce values whose type is a subtype of the function's parameter type.

To illustrate co-contextual type checking with subtypes, consider PCF with records and the usual depth and width subtyping for records. When type checking the expression $e = x.m + x.n$ with free x , we get the following derivation:

4. Simple Extensions of PCF

$$\begin{array}{c}
\text{T-ABS} \frac{e : T_2 \mid C \mid R \quad C_x = \{T_1 <: R(x) \mid \text{if } x \in \text{dom}(R)\}}{\lambda x : T_1. e : T_1 \rightarrow T_2 \mid C \cup C_x \mid R - x} \\
\\
\text{T-APP} \frac{e_1 : T_1 \mid C_1 \mid R_1 \quad e_2 : T_2 \mid C_2 \mid R_2 \quad U_1, U_2 \text{ is fresh} \quad \text{merge}(R_1, R_2) = R|_C}{e_1 e_2 : U_2 \mid C_1 \cup C_2 \cup \{T_1 = U_1 \rightarrow U_2, T_2 <: U_1\} \cup C \mid R} \\
\\
\text{T-IF0} \frac{e_1 : T_1 \mid C_1 \mid R_1 \quad e_2 : T_2 \mid C_2 \mid R_2 \quad e_3 : T_3 \mid C_3 \mid R_3 \quad \text{merge}(R_1, R_2, R_3) = R|_{C_R} \quad U \text{ is fresh} \quad C = \{T_1 = \text{Num}, U = T_2 \vee T_3\} \cup C_1 \cup C_2 \cup C_3 \cup C_R}{\text{if0 } e_1 e_2 e_3 : U \mid C \mid R} \\
\\
\text{T-FIX} \frac{e : T \mid C \mid R \quad U_1, U_2 \text{ is fresh}}{\text{fix } e : U_1 \mid C \cup \{T = U_1 \rightarrow U_2, U_2 <: U_1\} \mid R}
\end{array}$$

Figure 4.3.: Co-contextual type rules for PCF with subtyping.

$$\begin{array}{c}
+ : \text{Num} \mid \left\{ \begin{array}{l} U_3 = U_1.m, U_4 = U_2.n, \\ U_3 = \text{Num}, U_4 = \text{Num}, \\ U_5 <: U_1, U_5 <: U_2 \end{array} \right\} \mid x : U_5 \\
\swarrow \quad \searrow \\
.m : U_3 \mid \{U_3 = U_1.m\} \mid x : U_1 \quad .n : U_4 \mid \{U_4 = U_2.n\} \mid x : U_2 \\
\mid \quad \mid \\
x : U_1 \mid \emptyset \mid x : U_1 \quad x : U_2 \mid \emptyset \mid x : U_2
\end{array}$$

We can simplify the resulting constraint set by eliminating U_3 and U_4 to get $\{\text{Num} = U_1.m, \text{Num} = U_2.n, U_5 <: U_1, U_5 <: U_2\}$, where U_5 is the type required for x . Importantly, the type of x must be a subtype of U_1 and U_2 , which in turn must provide fields m and n respectively. Thus, the type U_5 of x must be a record type that at least provides fields m and n . Indeed, when we close above expression e as in $\lambda x : T. e$, type rule T-ABS yields another constraint $T <: U_5$. Accordingly, type checking succeeds for an annotation $T = \{m : \text{Num}, n : \text{Num}, o : \text{Num} \rightarrow \text{Num}\}$. But type checking correctly fails for an annotation $T = \{m : \text{Num}\}$, because $T <: U_2$ by transitivity such that $\text{Num} = U_2.n$ is not satisfiable.

Constraint system. In order to handle and solve the constraints introduced to support structural subtyping, our constraint solver relies on ideas from local type inference [PT00] and subtype-constraint simplification [Pot01]. That is, we keep track of the lower and

upper bounds on type variables and gradually refine them. Type variables have a polarity (covariant, contravariant or invariant), which determines whether to maximize or minimize the type of a variable. We transitively normalize newly generated subtyping constraints to subtyping constraints on type variables.

5. Co-contextual Let-Polymorphism

In this chapter, we present a complex extension to the co-contextual type checker of PCF by considering the language feature of let-polymorphism. Let-polymorphism represents a crucial language construct of functional languages and is widely used, e.g., in Haskell or ML. The advantage of let-polymorphism is that variables, which have a polymorphic type, can be used with different type bindings in different places of a program. This extension is very challenging for the co-contextual type checker, since variables can be of a ground type or a polymorphic type. In the co-contextual type checking the concrete knowledge about these variables is obtained only after correlating their declaration and usage.

In the following, we first give a brief overview of the contextual let-polymorphism. Next, we describe the structures introduced for co-contextual let-polymorphism, followed by the constraints added to co-contextual PCF. Then, we illustrate how the co-contextual type checker with let-polymorphism works via an example. We describe the systematic translation of the typing rules from contextual to co-contextual let-polymorphism, and give the typing rules for the latter. Finally, we have evaluated the co-contextual let-polymorphism in a setting that approaches to a fully-fledged functional language, by implementing the list library of Haskell.

5.1. Contextual Let-Polymorphism

Let us consider let-polymorphism used in SML [Uni05]. A let expression has: 1) the let binding, which consists of a variable and a bound expression; 2) the let body, where the variable can have different usages, which results in different types. Throughout this chapter, we call the variable in the let binding a program variable. To support the different usages of the program variable in the let body, its type has to be polymorphic. This polymorphic type is represented through the so-called type schema.

Let us illustrate the above concepts given the example below:

```
let x =  $\lambda a.a$  in
  let y = x 1 in (+ y y)
```

The expression in this example is called a nested let. The outer let has as let binding $x = \lambda a.a$, where x is the program variable and $\lambda a.a$ is the expression associated to it. The type of this expression defines the type of the program variable, which is a polymorphic type. Hence, the type schema corresponding to x is generated from the type of the expression $\lambda a.a$. The inner let ($let y = x 1 in (+ y y)$) is the body of the outer let. This inner let has as let binding $y = x 1$ and let body $(+ y y)$.

In SML, a type schema is of the form $typeSchema = (Type, List[typeVar])$. A type schema is a type along with a list of type variables ($typeVar$), which are part of that type

5. Co-contextual Let-Polymorphism

and can have different substitutions for different usages of the program variable. SML uses an auxiliary function *schema* to generate the type schemas. That is, it considers the type of the bound expression, then it accumulates the unsolved type variables of this type that do not appear anywhere else in the program. The resulting type schema is the type of the bound expression itself and the list of unsolved type variables.

A type schema can be *instantiated* for all the different usages of the program variable in the let body. Namely, each usage of the program variable in the body of let, generates a fresh type variable. Each of these type variables represents a distinct substitution. Implementations of the auxiliary function to find the unsolved type variables, functions *schema* and *instantiate* for SML are shown in the lecture notes [Uni05].

Let us illustrate type schemas, their generation and instantiation given the example above. The type of the program variable x is $X \rightarrow X$ (as the resulting type of the λ -abstraction), where X is a type variable. The schema generated from this type is $schema(X \rightarrow X) = typeSchema(X \rightarrow X, List(X))$. Next, we observe that x is used in the application that determines the type of the program variable y .

The variable x is used only once in the body and hence only instantiated once. The resulting type is $instantiate(typeSchema(X \rightarrow X, List(X))) = Y \rightarrow Y$, where Y is a fresh type variable. x is applied to a number of type Num . Therefore, we can deduce that $Y = Num$ and the instantiated type of x is $Num \rightarrow Num$. The resulting types of the application of x in y is Num . Therefore, the generalized type of y is $typeSchema(Num, List())$. The list of type variables is empty since Num is a ground type and does not contain type variables.

5.2. Co-Contextual Structures for Let-Polymorphism

We use our dualism technique to co-contextualize let-polymorphism. That is, the typing context is removed; instead we introduce context requirements, which are propagated bottom-up. Context requirements, as for co-contextual PCF, are sets of type bindings from variables to types $R = \{x : T\}$. Also, the merge and remove operations on the context requirements do not change from the ones presented for co-contextual PCF in Chapter 3.

To extend the co-contextual PCF with let-polymorphism, we add two new types: *partial type schemas* and *unification schemas*. The extension of co-contextual PCF with these types is shown below:

$$\begin{array}{ll}
 T ::= \dots & \\
 \quad | TSchema(T, List[U]) & \text{type schema} \\
 \quad | US & \text{unification schema}
 \end{array}$$

Partial type schemas are similar to SML type schemas and represent the polymorphic type of a program variable. We use the term partial, since they do not contain only type variables, but unification variables that can be substituted by ground types. Partial type schemas have the form $TSchema = (T, List[U])$, where T is a type and $List[U]$ is the list of all unification variables for the type T . The reason for using unification variables as part of the partial type schema is that the co-contextual type checking is performed

bottom-up and the type checker has no knowledge about the actual type of the bound expression containing variables. For example, when encountering the application $x \ 1$ the type of x is yet unknown and may be a ground type or a type variable. Therefore, the type schema we introduce is constructed from a type and the list of all unification variables of the type under scrutiny. This list represents unification variables that are placeholders for 1) type variables, or 2) ground types. In the latter case, those unification variables are removed from the list since they do not represent a polymorphic type.

Let us consider the above example. The co-contextual type checker starts at the leaves of the syntax tree and, while moving up the tree, gathers more information about the used variables. When encountering the declaration of y , the type checker does not know the type of x . It is known that x is applied to a number but the return type is unknown. Therefore, the type of x is $Num \rightarrow U_0$, where U_0 is a fresh unification variable. The resulting type of the application is U_0 . Therefore, the generalized type of y is $TSchema(U_0, List(U_0))$. When encountering the let binding of x , we learn that x is of type $U \rightarrow U$. Thus, the type of U_0 is actually a number (ground type). But at the point of type checking the let binding of y , this information is not yet provided. Therefore, unification variables are considered as type variables until proven otherwise.

Unification schemas are the second construct introduced for the co-contextual type checker. Since the type checker has no information on the actual types of the program variables used in expressions, we do not know their actual partial type schemas. For example, while type checking the body of the inner let from the example above, we do not know the actual type of the variable y . Therefore, we associate y to the so-called unification schema (US), as placeholder for its actual type, which is a partial type schema. We assume that all variables are program variables until proven otherwise. The US serves as placeholder for both cases, when a type is a partial type schema, or a non-polymorphic type. When encountering the let binding of y , these unification schemas will be resolved to a partial type schema ($US = TSchema(U_0, List(U_0))$).

In summary, we have introduced two new types, partial type schemas consisting of unification variables and unification schemas. Unification variables serve as placeholders only for non-polymorphic types while unification schemas can be used as placeholders for all types. Both constructs are necessary in the co-contextual type system and their usage during the type checking is discussed in the following section, where we will describe the co-contextual constraints of let-polymorphism in detail.

5.3. Co-Contextual Constraints for Let-Polymorphism

Previously, we talked about partial type schemas, but how are they generated and later instantiated in the body of let? We answer this question in the following of this section. Similar to SML, we first generalize the type of bound expression, which we assign to the program variable, and then instantiate the type of the program variable for all its usages in the let body. In order to perform these two steps in the co-contextual type system, we introduce *partial generalization constraints* and *instantiation constraints*. Both of these constraints are discussed in detail in the following subsections. In addition, we

5. Co-contextual Let-Polymorphism

introduce *substitution constraints* and *compatibility constraints*. A substitution constraint replaces unification variables \bar{U} , which occur in a type to other unification variables. These constraints are similar to the constraints introduced by parametric polymorphism in the previous Chapter 4 and, hence, are not discussed in more detail in this chapter. Compatibility constraints are used to correlate the program variable with its instances from the let body. Their usage is tied to the instantiation and both are described in detail in this section. In summary the set of constraints is extended as follows:

$c ::= \dots$	
$ T = GenP(T)$	partial generalization constraint
$ T = Inst(T)$	instantiation constraint
$ T = \{\bar{U} \mapsto \bar{U}\}T$	substitution constraint
$ T \sim T$	compatibility constraint

5.3.1. Partial Generalization Constraint

As dual to the generalization function schema found in the contextual let-polymorphism, we introduce partial generalization constraints using the function $GenP$ that generates partial type schemas. Unification variables of a type T are used as part of the list ($List[U]$) of the partial type schema, which is generated from generalizing T , as explained in previous section. We introduce an auxiliary function to extract the unification variables from a type, then perform the partial generalization to obtain a partial type schema. This function is called $occurUVar$ and is shown below:

```

occurUVar(t : Type): List[U] = {
  t match {
    case Num => List()
    case U => List(U)
    case t1 -> t2 => occurUVar(t1) ++ occurUVar(t2)
    case _ => List()
  }
}

```

We distinguish different cases for the type we want to generalize. First, if the type is a ground type, i.e., a number, then it does not have unification variables. Second, if the type is a unification variable, then it is added to the list. Finally if a type is a function type then we call the function recursively on both the argument type $t1$ and the result type $t2$. As a result the partial generalization function for obtaining partial type schemas is:

$$GenP(T) = TSchema(T, occurUVar(T))$$

Note that a schema generated from a ground type is considered to be ground, such that $TSchema(T, List()) = T$.

5.3.2. Instantiation Constraint

Instantiation constraints ($T = \text{Inst}(T)$) are the dual to the instantiate function described for contextual let-polymorphism. Inst is the partial instantiation function, which generates instances of co-contextual types. This function is applied to all let types, including the partial type schemas, as shown below:

$$\text{Inst}(T) = \begin{cases} T & \text{if } T \text{ is ground} \\ U & \text{if } T = U \\ \text{Inst}(T) \rightarrow \text{Inst}(T) & \text{if } T = T \rightarrow T \\ \{\bar{U} \mapsto \bar{U}'\}T \text{ where } \bar{U}' \text{ are fresh} & \text{if } T = T\text{Schema}(T, \bar{U}) \\ \text{Inst}(US) & \text{if } T = US \end{cases} \quad (5.1)$$

We distinguish five cases for partial instantiation. Ground types are already instances and the instantiation is the ground type itself. Likewise, unification variables are instances (in the let formalism) and instantiation is the unification variable itself. Function types recursively instantiate their parameter type and return type. Partial type schemas are instantiated by generating fresh unification variables for all usages of the program variable in the body of let. Finally, unification schemas cannot be instantiated, since they are only placeholders for yet to be determined types, i.e., all types in the previous four cases of instantiation are possible.

In contrast to the contextual instantiate function, the co-contextual type checking does not instantiate type variables but unification variables, since it is bottom-up with no information on the program variable's type. Instead we generate fresh unification variables, which correspond to the unification variables of a partial type schema. These unification variables are potential type variables or simply ground types.

The concept of instantiation is very challenging, since the type information must be correlated between the types of the instances encountered in the let body and type of the program variable in the let binding. The relation between the instances (fresh unification variables) and the instantiated type (unification variables that are part of the partial type schema) needs to be captured in the constraint system via compatibility constraints.

For example, let us consider the partial type schema of the program variable y from the example above ($T\text{Schema}(U_0, \text{List}(U_0))$). y is used two times in the body of let ($+ y y$), therefore there will be two instances of y . Each of the instances is associated to a fresh unification variable: U_1 and U_2 , respectively. Both U_1 and U_2 have type Num due to being operands of $+$. Let us now consider the let binding of y , which was associated with the unification variable U_0 . U_0 is the result type of the function application of x to the number 1. The program variable x represents an identity function with the type $U \rightarrow U$. Therefore, applying the identity function to a number gives as a result a number. Consequently, we get that U_0 is also a number ($U_0 = \text{Num}$). Since U_0 is a number, which is ground type, then its instances must be of the same ground type. The program is well-typed because U_1 and U_2 are also numbers, but the relation between U_0 , U_1 , and U_2 needs to be captured for this to be checked. If we consider a different program

5. Co-contextual Let-Polymorphism

instead, where x is applied to a character $'a$ of type $Char$, i.e., $y = x 'a$, the type of U_0 is $Char$. However, the instances of y are of type Num and the program is not well-typed ($Num = Char$). Without further constraints, U_0 , U_1 , and U_2 are distinct unification variables, with no connection among each other, and the type error would not be found.

We propose to generate compatibility constraints (of the form $T \sim T$) to correlate the program variable and its instances. These constraints are generated when encountering an instantiation constraint on a partial type schema. That is, when more information on the types is provided and the unification schemas are resolved to partial type schemas.

We show when this relation is generated and how the compatibility constraints operate in the following:

Assumption 1. *Given a set of constraints C and $(T = Inst(TSchema(\overline{U'}, List(U)))) \in C$, then applying $Inst$ yields $T = \{\overline{U} \mapsto \overline{U'}\}T$ and C is updated to $C \cup \{U \sim U'\}$, where $\overline{U'}$ are fresh unification variables.*

Compatibility constraints will be translated into equality constraints when additional type information about the program variable is provided. Given a program variable of type T_0 and instances in the let body of types T_1, \dots, T_n , then the set of compatibility constraints is of the form $\{T_0 \sim T_1 \dots T_0 \sim T_n\}$ and the following rules apply:

$$\frac{T_0 \sim T_1 \dots T_0 \sim T_n \quad T_0 \text{ is ground}}{T_0 = T_1 \dots T_0 = T_n}$$

$$\frac{T_0 \sim T_1 \dots T_0 \sim T_n \quad \overline{U} \text{ fresh} \quad T_0 \text{ is not ground}}{T_1 = U_1 \dots T_n = U_n}$$

We distinguish two cases for the type T_0 : 1) it is a ground type, or 2) a type variable (a unification variable is considered to be a type variable if we do not find a substitution for it). In the first case, all its instances should be of the same ground type. To ensure this, equality constraints are generated, i.e., $T_0 = T_1 \dots$. In the second case, a type variable can have instances of different types and these should not be equal to each other. Hence, we generate equality constraints between fresh unification variables and the instances of T_0 (one for each instance).

Let us consider the instantiations of the program variable y with type $TSchema(U_0, List(U_0))$. Instantiating a partial type schema generates compatibility constraints. As described above U_0 is instantiated two times U_1 and U_2 . Therefore, two compatibility constraints are added, i.e., $U_0 \sim U_1$ and $U_0 \sim U_2$.

For example in the initial expression where x is applied to a number, the unification variable U_0 is also a number. Consequently, compatibility constraints are turned to equality constraints indicating that also U_1 and U_2 must be numbers, i.e., $U_0 = U_1$, $U_0 = U_2$ and we know that $U_0 = Num$, therefore, $U_1 = Num$ and $U_2 = Num$. In contrast, when x is applied to a $Char$, then also $U_0 = Char$. Therefore, the compatibility constraints ($U_0 \sim U_1$ and $U_0 \sim U_2$) are turned to equality constraints indicating that U_1

and U_2 should both be *Char*, i.e., $U_1 = \text{Char}$ and $U_2 = \text{Char}$. But, U_1 and U_2 must be numbers due to being operands of $+$. Hence, we obtain the following constraints: $U_1 = \text{Char}$, $U_2 = \text{Char}$, $U_1 = \text{Num}$ and $U_2 = \text{Num}$. The constraint solver will deduce that $\text{Num} = \text{Char}$, which does not hold. Therefore, the program is not well-typed.

5.3.3. Continuous Solving of Constraints

We optimize the constraint solving w.r.t. the number of constraints propagated to the root of the syntax tree via continuous solving of constraints. Namely, constraints are solved as early as possible, i.e., in the moment the necessary information is present. While performing continuous solving of constraints, we can apply substitution to instantiation constraints for each solved constraint. If the instantiated type is a partial type schema after the substitution, then compatibility constraints are generated. When we know more information on the unification variables, we can continue applying the substitution to the compatibility constraints. For example, let us suppose $(U_0 \sim U_1$ and $U_0 \sim U_2)$ and we learn that U_0 is ground at the current state of type checking, then equality constraints between U_0 and $U_1 \dots U_n$ are generated.

This enables the constraint solver to solve the compatibility constraints and generate equality constraints as early as possible, and not to leave everything for the end of type checking, where more constraints are collected and would take longer time to solve.

5.4. Co-Contextual Let-Polymorphism by Example

In this section, we describe the co-contextual type checking of let-polymorphism in a step-by-step manner. As a reminder, while type checking an expression, we denote each step as $e : T | R$, where T the type of the expression e and R shows the generated requirements. The illustrate the steps of the type checker we consider the example given in Section 5.1.

$$\begin{aligned} &\text{let } x = \lambda \text{ a.a in} \\ &\quad \text{let } y = x \ 1 \text{ in } (+ \ y \ y) \end{aligned}$$

Type checking starts at the body of the nested let $(+ \ y \ y)$ without any information on the variables used in the expressions. The co-contextual type checker does not know the types of the variables y and therefore cannot infer the type of the $+$ operation. Each of these expressions is assigned to a fresh unification schema $y : \text{Inst}(US_0) | \{y : US_0\}$, $+$: $\text{Inst}(US_1) | \{+ : US_1\}$, $y : \text{Inst}(US'_0) | \{y : US'_0\}$. Since y is the same variable, we merge the two requirements on y , which generates the constraint $US_0 = US'_0$. As a result US'_0 is discarded and the refined type of $+$ is $\text{Inst}(US_1) = \text{Inst}(US_0) \rightarrow \text{Inst}(US_0) \rightarrow U$, where U is a fresh unification variable.

The type checker continues with the nested let's binding $(y = x \ 1)$. First, the expression $(x \ 1)$ is determined via $x : \text{Inst}(US_2) | \{x : US_2\}$, $1 : \text{Num} | \emptyset$, which yields that type $\text{Inst}(US_2) = \text{Num} \rightarrow U_0$, where U_0 is fresh. Second, the type of the program variable y is generalized and associated to its unification schema, i.e., $US_0 = \text{GenP}(U_0)$, which generates a partial type schema $US_0 = \text{GenP}(U_0) = \text{TSchema}(U_0, \text{List}(U_0))$.

5. Co-contextual Let-Polymorphism

Before moving to the next expression, the type checker replaces the type schema US_0 in the other constraints. For each instance in the body of let the type checker generates fresh unification variables, i.e., for each $y : Inst(US_0)|\{y : US_0\}$. The resulting types are $Inst(US_0) = Inst(TSchema(U_0, List(U_0))) = U_1$ and $Inst(US_0) = U_2$. Meanwhile, compatibility constraints are generated for each instance as follows $U_0 \sim U_1$ and $U_0 \sim U_2$. In addition, the type of $+$ is now refined to $Inst(U_1) = U_1 \rightarrow U_2 \rightarrow U$.

Next, the type checker considers the let binding $x = \lambda a.a$. The variable a is type checked, which gives $a : Inst(US_3)|\{a : US_3\}$. Consequently, the type of x is now refined to $US_2 = US_3 \rightarrow Inst(US_3)$. In a function, argument types cannot be polymorphic types; US_3 is placeholder for a partial type schema. Therefore, the type checker introduces a fresh unification variable U_3 , such that $US_3 = U_3$ and $Inst(U_3) = U_3$. The resulting type of the lambda abstraction is now $U_3 \rightarrow U_3$. Then, the type of the program variable x is generalized. $US_2 = GenP(U_3 \rightarrow U_3) = TSchema(U_3 \rightarrow U_3, List(U_3))$.

As before, the type checker replaces occurrence of US_2 to the generalized type. That is, $Inst(US_2) = Inst(TSchema(U_3 \rightarrow U_3, List(U_3))) = U_4 \rightarrow U_4$, and the compatibility constraint $U_3 \sim U_4$ is generated. At this point the type checker can already solve $U_4 \rightarrow U_4 = Num \rightarrow U_0$ to $U_4 = Num$ and $U_0 = Num$. Since U_0 is used in the compatibility constraints and is known to be a ground type, the respective compatibility constraints can be turned to equality constraints. That is, $U_0 = U_1 = U_2 = Num$. U_3 is not ground, so it remains a type variable.

At the point in time when the type checker encounters the type signature of $+$, the whole expression can be checked. That is, we learn that $+$ takes two *Nums* as arguments and returns a *Num*. From this, we can deduce that $U_1 \rightarrow U_2 \rightarrow U = Num \rightarrow Num \rightarrow Num$, that is $U_1 = Num$, $U_2 = Num$, $U = Num$. In the end, this program is well-typed because we already deduced that U_1 , U_2 are *Num* and $Num = Num$ holds.

5.5. Co-Contextual Typing Rules

In this section, we describe the co-contextual typing rules of let-polymorphism. We extend the rules for variable typing, lambda abstraction and application, to incorporate the additional type schemas and unification variables required in co-contextual let-polymorphism. Finally, we added a new rule for the let expression. The complete set of rules is presented in Figure 5.1, below.

Variable Typing

In contrast to PCF, in let-polymorphism while type checking a variable (T-PVAR), it is assumed that all variables are potential program variables. But, the type of x cannot be determined in lack of the typing context. Therefore, the type checker generates a fresh unification schema, which is associated to the variable x and added to the requirements set. The result of type checking the variable x should be an instantiation of the actual type of x . Since the type of x is not known, the result of instantiating this type is also unknown. Therefore, we generate a fresh unification variable U , such that $U = Inst(US)$,

$\text{T-PVAR} \frac{US, U \text{ are fresh}}{x : U \mid U = \text{Inst}(US) \mid x : US}$	$\text{T-PABS} \frac{\begin{array}{l} e : T \mid C \mid R \quad U \text{ is fresh} \\ C_x = \{U = R(x) \mid \text{if } x \in \text{dom}(R)\} \end{array}}{\lambda x. e : U \rightarrow T \mid C \cup C_x \mid R - x}$
$\text{T-PApP} \frac{\begin{array}{l} e_1 : T_1 \mid C_1 \mid R_1 \quad e_2 : T_2 \mid C_2 \mid R_2 \\ U, U_1 \text{ are fresh} \quad \text{merge}_R(R_1, R_2) = R _C \end{array}}{e_1 e_2 : U \mid C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow U\} \cup \{U_1 = T_2\} \cup C \mid R}$	
$\text{T-LETPOLY} \frac{\begin{array}{l} e_1 : T_1 \mid C_1 \mid R_1 \quad e_2 : T_2 \mid C_2 \mid R_2 \quad \text{merge}_R(R_1, R_2 - x) = R _C \\ C_g = \{R_2(x) = \text{GenP}(T_1) \mid \text{if } x \in \text{dom}(R_2)\} \end{array}}{\text{Let } x = e_1 \text{ in } e_2 : T_2 \mid C_1 \cup C_2 \cup C \cup C_g \mid R}$	

Figure 5.1.: A co-contextual constraint-based formulation of the type system of PCF with Let-Polymorphism.

which is added to the constraint set. The resulting type from type checking the variable x is then U .

Lambda Abstraction Typing

The rule T-PABS is used to type check lambda abstractions, which are unannotated. In this case, we generate a fresh unification variable U as placeholder for the actual type of the variable x . We do not generate a unification schema, because there is no polymorphism inside a function. If there is a requirement for x , we add the constraint $U = R(x)$ and remove that requirement $R - x$. The resulting type of this expression is the function type $U \rightarrow T$, where T is the type of e .

Application Typing

The first expression e_1 in the rule T-PApP has a function type. Therefore, the return type of this function is a fresh unification variable (U) and not a unification schema. In addition, we want to ensure that the type of e_2 (as part of the function type) is a non-polymorphic type. Therefore, we generate another fresh unification variable U_1 and add the constraint $U_1 = T_2$.

Polymorphic Let Typing

Finally, we consider the rule T-LETPOLY. The bound expression e_1 and the body of let (e_2) are type checked. The type of x is a generalization of the type of e_1 (T_1). Then, if the program variable x has usages in the body of let, a requirement for x should exist in R_2 .

5. Co-contextual Let-Polymorphism

If there is a requirement for x , we add the constraint $R_2(x) = \text{GenP}(T_1)$ and remove that requirement $R_2 - x$.

5.6. Implementation and Evaluation

We have implemented the co-contextual let-polymorphism in Scala, as an extension to our previous implementation of co-contextual PCF. All typing rules regarding let-polymorphism are implemented in the co-contextual version. Also, we implemented the required types to support let-polymorphism: *TSchema* and *US*. As part of the constraint system, we added generalization, instantiation, and compatibility constraints. We implemented the continuous solving of constraints and added an additional optimization for compacting compatibility constraints.

The optimization for compatibility uses a compact list form, since all compatibility constraints relate one element to multiple others, i.e. $T_0 \sim T_1$, $T_0 \sim T_2$, \dots , and can be replaced by $T_0 \sim [T_1, T_2, \dots]$. This optimized form of constraints is stored outside the set of constraints that are considered for continuous solving. As such the optimization serves a dual purpose, by a) reducing the size of data structures, and b) also preventing the constraint solver from processing compatibility constraints, which will only be solvable during substitution to ground types.

We have evaluated our approach by writing a large variety of both well-typed and not well-typed programs using let expressions. These programs range from simple expressions as exemplified in this chapter to complex programs that deal with non-intuitive corner cases. In all cases we positively identified all type errors and ensured that the typing is correct. In addition to these programs, we have evaluated the co-contextual let-polymorphism in a setting that approaches to a fully-fledged functional language, namely Haskell. We have implemented the list library of Haskell and written programs using this library. We implemented the most important operations on lists like *append*, *head*, *tail*, *last*, *init*, *length*, *map*, etc. In order to implement these operations, we have looked into the Haskell list library source code¹ and implement the co-contextual version of them.

In order to support the Haskell list library, we added new types for lists and empty lists, respectively *List*[*T*], *List*[], We added co-contextual typing rules for recursive let *T-LETREC* and pattern matching on lists *T-MATCH*, as shown in Figure 5.2. The typing rule for pattern matching is straightforward, so we will explain the rule *T-LETREC* in more detail. In contrast to *T-LETPOLY*, *T-LETREC* allows variables, like x to have a recursive definition. Namely, x can appear also in the expression e_1 . Consequently, there could be a requirement regarding x in R_1 . In this case, inside the let binding, x does not have a polymorphic type. Hence, the type of x in R_1 is equal to the type of e_1 and not a generalization of it ($R_1(x) = T_1$), as part of the recursive definition. The requirement for x in R_1 is satisfied since we know the actual type T_1 of x and is removed $R_1 - x$. Next, if there is a requirement for x in R_2 , its type is a generalization of T_1 ($R_2(x) = \text{GenP}(T_1)$) and we remove that requirement $R_2 - x$.

¹<http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-List.html>

$$\begin{array}{c}
\begin{array}{c}
e_1 : T_1 \mid C_1 \mid R_1 \quad e_2 : T_2 \mid C_2 \mid R_2 \\
C_m = \{R_1(x) = T_1 \mid \text{if } x \in \text{dom}(R_1)\} \\
C_g = \{R_2(x) = \text{GenP}(T_1) \mid \text{if } x \in \text{dom}(R_2)\} \\
\text{merge}_R(R_1 - x, R_2 - x) = R|_C
\end{array} \\
\text{T-LETREC} \frac{}{\text{Let } x = e_1 \text{ in } e_2 : T_2 \mid C_1 \cup C_2 \cup C \cup C_g \mid R} \\
\\
\begin{array}{c}
e_1 : T \mid C_1 \mid R_1 \quad e_2 : T_r \mid C_2 \mid R_2 \quad e_3 : T' \rightarrow \text{List}[T'] \rightarrow T'_r \mid C_3 \mid R_3 \\
C_r = \{(\text{List}[T] = \text{List}[T']) \cup (T_r = T'_r)\} \quad \text{merge}_R(R_1, R_2, R_3) = R|_C
\end{array} \\
\text{T-MATCH} \frac{}{\text{match } e_1 \ e_2 \ e_3 : T_r \mid C_1 \cup C_2 \cup C_3 \cup C_r \cup C \mid R}
\end{array}$$

Figure 5.2.: A co-contextual constraint-based formulation of the type system of PCF with Let-Polymorphism and Lists.

With the above additions of let-polymorphism and Lists the co-contextual type checker was powerful enough to type-check programs using Haskell List constructs and their accompanying operations. We have written a selective set of programs that utilize all implemented operations of the co-contextual list library. As before we have written well-typed and not well-typed programs. The co-contextual type checker has positively identified all type errors and correctly typed the programs, bringing it into the range of a fully-fledged functional programming language.

6. Related Work

In this chapter, we discuss work related to our co-contextual formulation of type rules for functional languages and work related to incremental type checking.

Co-contextual type rules are different from type inference, because type inference relies on the context to coordinate the types of variables [Mil]. That is, type inference assigns the same type to all references of the same bound variable. In contrast, co-contextual type checking assigns each variable reference a fresh type variable and coordinates them through requirement merging.

Our co-contextual formulation of type rules is related to prior work on principal typing [Jim96, Wel02], not to be confused with principal types. A principal typing of some expression is a derivation that subsumes all other derivations. Specifically, a principal typing (if it exists) requires the weakest context and provides the strongest type possible for the given expression. Principal typing finds application in type inference where, similar to our work, the minimal context requirements can be inferred automatically. We extend the work on principal typing by identifying the duality between contexts and context requirements as a method for systematically constructing co-contextual type systems. Such a method has not been formulated in previous work. Moreover, prior work on principal typing only considered ad-hoc incrementalization for type checking top-level definitions, whereas we describe a method for efficient fine-grained incremental type checking.

Other formulations of type systems related to our co-contextual one have also been used in the context of compositional compilation [ADDZ05] and the compositional explanation of type errors [Chi01]. However, these systems only partially eliminated the context. In the work on compositional compilation [ADDZ05], type checking within a module uses a standard contextual formulation that coordinates the types of parameters and object self-references `this`. For references to other modules, the type checker generates constraints that are resolved by the linker. Using our method for constructing co-contextual type systems, we can generalize this type system to eliminate the local context as well, thus enabling compositional compilation for individual methods. In the work on compositional explanations of type errors [Chi01], only the context for monomorphic variables is eliminated, whereas a context for polymorphic variables is propagated top-down. Our extension for parametric polymorphism demonstrated that our co-contextual formulation of type rules can support polymorphic variables and eliminate the context entirely.

Snelting, Henhapl, and Bahlke’s work on PSG and context relations [SH86, BS86, Sne91] supports incremental analyses beyond traditional type checking and provide a join mechanism that is similar to our merge operation. However, context relations are very different from our proposal both conceptually as well as technically. Conceptually, our

6. Related Work

main conceptual finding is the duality between contexts and co-contexts that informs the design of co-contextual type checkers. For example, we used this duality to essentially derive bottom-up type checkers that support subtyping and polymorphism. In contrast, it is not obvious how to extend context relations and their unification-based approach to support subtyping (beyond enumerating subtyping for base types like `int` or `float`) or user-defined polymorphism (with explicit type application). To use context relations, the user has to come up with clever encodings using functionally augmented terms. The duality we found provides a principle for systematically identifying appropriate encodings. Technically, we do not use relations with cross-references to represent analysis results and we do not rely on a separate name resolution that runs before type checking. Instead, we use user-supplied constraint systems and context requirements. In particular, this enables us to solve constraints locally or globally and to apply solutions eagerly or propagate solutions up the tree (with performance impacts as shown in Section 3.4).

Kahn, Despeyroux, et al.’s work on Typol [Des84, Kah87] compiles inference rules to Prolog. Thus, Typol benefits from Prolog’s support for solving predicates such as `tcheck(C,E,T)` for any of the variables. In contrast, our main contribution is to systematically eliminate the context through duality. In particular, the practicality of using Prolog to infer context requirements is not documented, and it is not clear if the minimal context is always found. Attali et al. present an incremental evaluator for a large class of Typol programs including type checkers [ACG92]. However, their incrementalization requires that the context is stable and does not change. If the context changes, the previous computation is discarded and gets repeated.

Meertens describes an incremental type checker for the programming language B [Mee83]; it collects fine-grained type requirements, but is not clear on requirement merging and also does not solve type requirements incrementally. Johnson and Walz describe a contextual type checker that incrementally normalizes type constraints while type checking a program in order to precisely identify the cause of type errors [JW86]. Aditya and Nikhil describe an incremental Hindley/Milner type system that only supports incremental rechecking of top-level definitions [AN91]. Miao and Siek describe a type checker for multi-staged programming [MS10]; type checking in later stages is incremental with respect to earlier stages, relying on the fact that types only get refined by staging but assumptions on the type of an expression never have to be revoked.

Wachsmuth et al. describe a task engine for name resolution and type checking [WKV⁺13]. After a source file changes, the incremental type checker generates new tasks for this file. The task engine reuses cached task results where possible, computes tasks that have not been seen before, and invalidates cached results that depend on tasks that have been removed. The type checker relies on an incremental name analysis, whose tasks are incrementally maintained by the same engine.

Eclipse’s JDT performs incremental compilation through fine-grained dependency tracking.¹ It evolved from IBM’s VisualAge [CLR98], which stores the code and configuration of a software project in an in-memory database with separate entries for different artifacts, such as methods, variables, or classpaths. The system tracks dependencies between indi-

¹<http://eclipse.org/jdt/core/>

vidual artifacts in order to update the compilation incrementally. Similarly, Facebook’s Flow² and Hack³ languages feature incremental type checking through a background server that watches source files in order to invalidate the type of changed files and files that are affected by them. Unfortunately, not much information on the incremental type checkers of Eclipse and Facebook is available, and it is not clear how to construct similar incremental type checkers systematically.

Acar et al. [ABH02] present a mechanism for adaptive computation based on the idea of modifiable references and three operations for creating (*mod*), reading (*read*), and writing (*write*) modifiabiles. This mechanism can dynamically create new computations and delete old computations. To illustrate their mechanism, they change a small ML library. They record the value of an expression that may change from changes to the inputs. To incrementalize a program, the programmer 1) identifies locations whose changes should trigger a recomputation; and 2) writes functions that carry out the incremental update on these locations. The main contribution is the change-propagation algorithm based on these modifiabiles. Carlsson [Car02] changed the ML library of Acar et al [ABH02] into purely functional Haskell. They did so by adding support for *Monads*. In the end they provided a monadic framework that can be used as an interface to an imperative library for incremental computations.

Liu et al. [LT95] describe how to derive incremental programs from non-incremental ones, combining a number of program analysis and transformation. This work supports also let-polymorphism. This is an extended version of the work on lambda calculus by Field et al. [FT90]. They do systematic changes on the computations to support the so-called information and cache sets. Informations sets of subexpressions are collected, then the cache set is extended with these information sets. These are used to simplify the subexpressions and replace subexpressions whose values can be efficiently retrieved from the cached sets. In contrast to our work, they do not change type systems of the underlying programs, rather than finding efficient ways to better understand incremental computations. Their aim is to establish a general framework on incremental computations.

On the other hand, there are general-purpose engines for incremental computations. Before implementing incremental type checkers directly, we experimented with an implementation of co-contextual type rules based on an incremental SQL-like language developed by Mitschke et al. [MEK⁺14]. In our experiment, the overhead of general-purpose incrementalization was significant and too large for the fine-grained incremental type checking that we are targeting. For this reason, we did not further consider other general-purpose engines for incremental computations, such as attribute grammars [DRT81] or self-adjusting computations [ABB⁺09].

Our implementation strategy is similar to the general-purpose incrementalization system Adapton [HPHF14]. Like Adapton, after a program (data) modification, we also only rerun the type checker (computation) when its result is explicitly required again. This way, multiple program modifications can accumulate and be handled by a single type-check run. In contrast to Adapton, we also propagate dirty flags only on-demand.

²<http://flowtype.org>

³<http://hacklang.org>

7. Part Summary

In this part, we described the initial idea of co-contextual type checking and considered a relatively simple type system, as PCF. We use dualism as a technique to systematically translate constructs of contextual type systems to co-contextual constructs. In doing so, the top-down propagated context is replaced by the bottom-up propagated context requirements. These requirements are generated independently for different subexpressions of an expression. Therefore, the co-contextual type checker decouples the dependencies between subexpressions. Context requirements consist of a set of bindings from variables to fresh unification variables. These requirements are satisfied when the type checker encounters the actual types of the required variables. In addition to requirements, co-contextual type systems use constraints to resolve variables to their actual types. Requirements can be merged and removed and in both cases constraints are generated. A program is well-typed in a co-contextual setting if all requirements are satisfied and all constraints are solved.

Operations on the requirements are dual to the operations on the typing context. For example, for each variable added to the typing context, the requirement corresponding to it is removed because the actual type of that variable is provided. We gave a detailed explanation of all operations on the context requirements. Later, we described the translation from contextual to co-contextual typing rules. The translation is systematically applying the dualism technique to all rules of the contextual type system. Next, we proved the equivalence between the contextual and co-contextual type systems. The type of an expression is the same in both contextual and co-contextual type checkers. That is, if an expression is well-typed given a contextual type checker then it is also well-typed in a co-contextual setting. Next, we showed that a co-contextual type checker facilitates incremental type checking because it removes the dependencies between subexpression. We use memoization as technique to feature incrementalization. Performance is further improved by enabling incremental solving of constraints and eager substitution. The incremental type checker performs better by an order of magnitude compared to the non-incremental and contextual type checkers.

To further evaluate our dualism technique and evaluate which changes are required in the presence of different language constructs, we introduced extensions of PCF and co-contextualized their type systems. We considered records, parametric polymorphism and structural subtyping, as comparatively simple extensions. We conducted systematic changes to all constructs of the co-contextual PCF in order to support each extension. First, types and expressions were extended. For example, in the case of records, we added record type and record expression. Second, we extended the set of constraints. For example, in the case of parametric polymorphism, we added universal-type substitution constraints. Furthermore, we introduced universally quantified unification variables,

7. Part Summary

e.g., $\forall U.T$, which binds the type variable that will replace the unification variable U . Third, we extended the set of requirements. For example, in parametric polymorphism, new type-variable requirements were added to co-contextualize type-variable handling. Furthermore, the operations on requirements were adapted, in the presence of structural subtyping, since subtyping weakens the assumptions about variable bindings. That is, in the presence of subtyping values must adhere to bounded types, which affects the merging of context requirements.

Finally, we have introduced let-polymorphism, as powerful language construct, that brings the co-contextual type system closer to a full-fledged functional language such as Haskell. Let-polymorphism is challenging to add to PCF because variables can have different bindings in different parts of the program. In order to feature the different bindings and co-contextualize let-polymorphism, we have introduced partial type schemas as a dual concept to contextual type schemas. Since the information on a program variable's actual type is not known up-front due to the bottom-up type checking, we introduced unification type schemas as placeholders that are concretized later with partial type schemas, or non-polymorphic types. In addition, we introduced generalization and instantiation constraints. While applying the generalization constraint, we do not have full knowledge of a type and its unification variables. Hence, a unification variable that is considered as type variable of a schema and is instantiated, could be substituted to a ground type. This rises errors because the instances of the unification variable could have different types from its actual ground type. These errors are captured by introducing compatibility constraints, which keep track of the relations between the unification variables and their instances. We have evaluated our extension, by implementing the Haskell List library. A minimal extension to the co-contextual type system was required to introduce recursive data types, i.e., Lists. With the addition of let-polymorphism the co-contextual type checker was powerful enough to type-check programs using Haskell List constructs.

Part III.

Co-Contextual Type Checkers for Object-Oriented Languages

Part III Overview

In this part, we describe how to construct co-contextual type checkers for object-oriented (OO) languages. We consider Featherweight Java (FJ), as a core language of OO. In our co-contextual setting, we first cover the following key features of OO languages: subtype polymorphism, nominal typing and inheritance. Contextual type checkers encode these features in the form of class tables, an additional kind of typing context inhibiting incrementalization. Therefore, we remove these class tables and introduce the dual structures of class table requirements. We describe the operations on class table requirements, and the co-contextual typing rules for FJ. We prove the equivalence of FJ's type system and our co-contextual formulation. Based on our co-contextual formulation of the FJ type system, we implement an efficient incremental type checker for FJ Java. We evaluate the performance of our type checker using both pure FJ programs as well as programs translated from Java (using only FJ as a subset), which allows us to make comparisons even with the standard Java type checker.

In addition to the key features of OO languages, we consider two non-trivial extensions: generics and method overloading. Extending the co-contextual FJ with these features is very challenging. Generics give rise to new types where a single generic declaration is tied to multiple instances that have different types depending on their usage. Method overloading allows multiple related declarations, and has the effect that a single usage must be correlated to the correct declaration, which is particularly hard in conjunction with subtyping. We extend the co-contextual FJ with both language features and explain the required changes.

8. Background and Motivation

In the following two sections, we first present the contextual FJ typing rules from [IK01] and then give an example to illustrate how contextual and co-contextual FJ type checkers work. The example provides a detailed illustration for the traditional class table context in comparison to the dual class table requirements.

8.1. Featherweight Java: Syntax and Typing Rules

Featherweight Java [IK01] is a minimal core language for modeling Java's type system. Figure 8.1 shows the syntax of classes, constructors, methods, expressions, and typing contexts. Metavariables C , D , and E denote class names and types; f denotes fields; m denotes method names; **this** denotes the reference to the current object. As is customary, an overline denotes a sequence in the metalanguage. Moreover, $\overline{C} \overline{f}$ is written as shorthand for $C_1 f_1, \dots, C_n f_n$, where n is the length of \overline{C} and \overline{f} , and $\overline{C} <: \overline{D}$ as shorthand for $C_1 <: D_1, \dots, C_n <: D_n$. Γ is a set of bindings from variables and **this** to types.

$L ::= \text{class } C \text{ extends } D \{ \overline{C} \overline{f}; K \overline{M} \}$	class declaration
$K ::= C(\overline{C} \overline{f}) \{ \text{super}(\overline{f}); \text{this}.\overline{f} = \overline{f} \}$	constructor
$M ::= C m(\overline{C} \overline{x}) \{ \text{return } e; \}$	method declaration
$e ::= x \mid \text{this} \mid e.f \mid e.m(\overline{e}) \mid \text{new } C(\overline{e}) \mid (C)e$	expression
$\Gamma ::= \emptyset \mid \Gamma; x : C \mid \Gamma; \text{this} : C$	typing contexts

Figure 8.1.: Featherweight Java syntax and typing context.

The type system, shown in Figure 8.2, ensures that variables, field access, method invocation, constructor calls, casting, and method and class declarations are well-typed. The typing judgment for expressions has the form $\Gamma; CT \vdash e : C$, where Γ denotes the typing context, CT the class table, e the expression under analysis, and C the type of e . The typing judgment for methods has the form $C; CT \vdash M \text{ OK}$ and for classes $CT \vdash L \text{ OK}$.

In contrast to the FJ paper [IK01], we added some cosmetic changes to the presentation. For example, the class table CT is an implicit global definition in FJ. Our presentation explicitly propagates CT top-down along with the typing context. Another difference to Igarashi et al. is in the rule T_{NEW} : Looking up all fields of a class returns a constructor signature, i.e., $\text{fields}(C, CT) = C.\text{init}(\overline{D})$ instead of returning a list of fields with their corresponding types. We made this subtle change because it clearer communicates the intention of checking the constructor arguments against the declared parameter types. Later on, these changes pay off, because they enable a systematic translation of typing

8. Background and Motivation

$\text{T-VAR} \frac{\Gamma(x) = C}{\Gamma; CT \vdash x : C}$	$\text{T-FIELD} \frac{\Gamma; CT \vdash e : C_e \quad \text{field}(f_i, C_e, CT) = C_i}{\Gamma; CT \vdash e.f_i : C_i}$
$\text{T-INVK} \frac{\Gamma; CT \vdash e : C_e \quad \Gamma; CT \vdash \bar{e} : \bar{C} \quad \text{mtype}(m, C_e, CT) = \bar{D} \rightarrow C \quad \bar{C} <: \bar{D}}{\Gamma; CT \vdash e.m(\bar{e}) : C}$	
$\text{T-NEW} \frac{\Gamma; CT \vdash \bar{e} : \bar{C} \quad \text{fields}(C, CT) = C.\text{init}(\bar{D}) \quad \bar{C} <: \bar{D}}{\Gamma; CT \vdash \text{new } C(\bar{e}) : C}$	
$\text{T-UCAST} \frac{\Gamma; CT \vdash e : D \quad D <: C}{\Gamma; CT \vdash (C)e : C}$	$\text{T-DCAST} \frac{\Gamma; CT \vdash e : D \quad C <: D \quad C \neq D}{\Gamma; CT \vdash (C)e : C}$
$\text{T-SCAST} \frac{\Gamma; CT \vdash e : D \quad C \not<: D \quad D \not<: C}{\Gamma; CT \vdash (C)e : C}$	
$\text{T-METHOD} \frac{\begin{array}{c} \bar{x} : \bar{C}; \text{this} : C; CT \vdash e : E_0 \quad E_0 <: C_0 \\ \text{extends}(C, CT) = D \\ \text{if } \text{mtype}(m, D, CT) = \bar{D} \rightarrow D_0, \text{ then } \bar{C} = \bar{D}; C_0 = D_0 \end{array}}{C; CT \vdash C_0 \ m(\bar{C} \ \bar{x})\{\text{return } e\} \text{ OK}}$	
$\text{T-CLASS} \frac{\begin{array}{c} K = C(\bar{D}' \ \bar{g}, \bar{C}' \ \bar{f})\{\text{super}(\bar{g}); \text{this}.\bar{f} = \bar{f}\} \quad \text{fields}(D, CT) = D.\text{init}(\bar{D}') \\ C; CT \vdash \bar{M} \text{ OK} \end{array}}{CT \vdash \text{class } C \text{ extends } D \ \{\bar{C} \ \bar{f}; K \ \bar{M}\} \text{ OK}}$	
$\text{T-PROGRAM} \frac{CT = \bigcup_{L' \in \bar{L}} (\text{addExt}(L') \cup \text{addCtor}(L') \cup \text{addFs}(L') \cup \text{addMs}(L')) \quad (CT \vdash L' \text{ OK})_{L' \in \bar{L}}}{\bar{L} \text{ OK}}$	

Figure 8.2.: Typing rules of Featherweight Java.

rules to co-contextual FJ (Sections 9.1 and 9.2) and give a strong and rigorous equivalence result for the two type systems (Section 9.3).

Furthermore, we explicitly include a typing rule T-PROGRAM for programs, which is implicit in Igarashi et al.'s presentation. The typing judgment for programs has the form $\bar{L} \text{ OK}$: A program is well-typed if all class declarations are well-typed. The auxiliary functions addExt , addCtor , addFs , and addMs extract the supertype, constructor, field and method declarations from a class declaration into entries for the class table. Initially, the class table is empty, then it is gradually extended with information from every class declaration by using the above-mentioned auxiliary functions. This is to emphasize

that we view the class table as an additional form of typing context, having its own set of extension operations. We describe the class table extension operations and their co-contextual duals formally in Section 9.1.

8.2. Contextual and Co-Contextual Featherweight Java by Example

We revisit the example from the introduction (Part I) to illustrate that, in absence of context information, maintaining requirements on class members is non-trivial:

`new List().add(1).size() + new LinkedList().add(2).size().`

```
class List extends Object {
  Int size() {...}
  List add(Int a){...}
}
class LinkedList extends List { }
```

Here, we assume the class declarations on the right-hand side: `List` with methods `add()` and `size()` and `LinkedList` inheriting from `List`. As before, we assume there are typing rules for numeric `Int` literals and the `+` operator over `Int` values. We use `LList` instead of `LinkedList` in Figure 8.3 for space reasons.

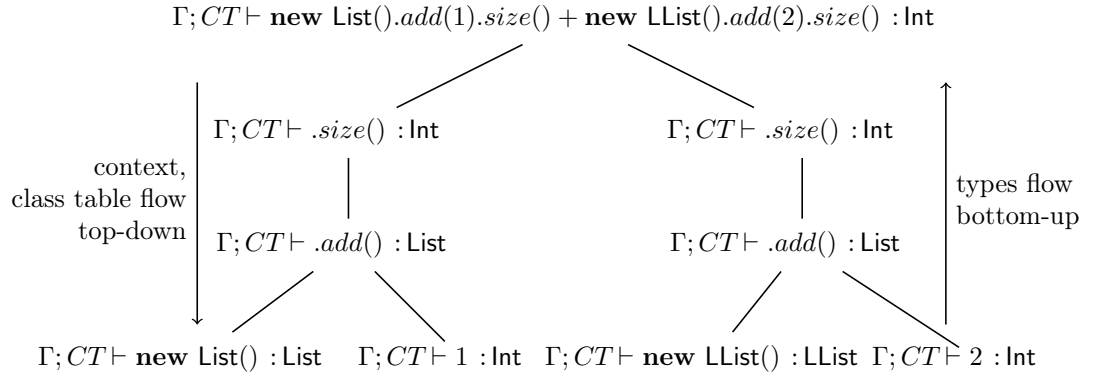
Figure 8.3 (a) depicts standard type checking with typing contexts in FJ. The type checker in FJ visits the syntax tree “down-up”, starting at the root. Its inputs (propagated downwards) are the context Γ , class table CT , and the current subexpression e . Its output (propagated upwards) is the type C of the current subexpression. The output is computed according to the currently applicable typing rule, which is determined by the shape of the current subexpression. The class table used by the standard type checker contains classes `List` and `LinkedList` shown above. The type checker retrieves the signatures for the method invocations of `add` and `size` from the class table CT .

To recap, while type checking constructor calls, method invocations, and field accesses the context and the class table flow top-down; types of fields/methods are looked up in the class table.

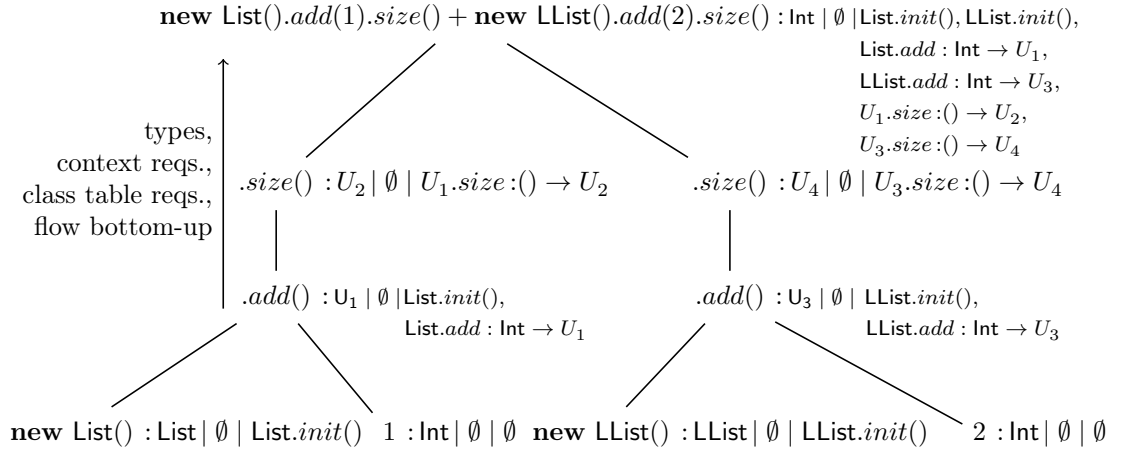
Figure 8.3 (b) depicts type checking of the same expression in co-contextual FJ. Here, the type checker starts at the leaves of the tree with no information about the context or the class table. The expression type T , the context requirements R , and class table requirements CR all are outputs and only the current expression e is input to the type checker, making the type checker context-independent. At the leaves, we do not know the signature of the constructors of `List` and `LinkedList`. Therefore, we generate requirements for the constructor calls `List.init()` and `LinkedList.init()` and propagate them as class table requirements. For each method invocation of `add` and `size` in the tree, we generate requirements on the receiver type and propagate them together with the requirements of the subexpressions.

In addition to generating requirements and propagating them upwards as shown in Figure 8.3 (b), a co-contextual type checker also *merges requirements* when they have compatible receiver types. In our example, we have two requirements for method `add` and two requirements for method `size`. The requirements for method `add` have incompatible ground receiver types and therefore cannot be merged. The requirements for method `size` both have placeholder receivers and therefore cannot be merged just yet. However, for

8. Background and Motivation



(a) Contextual type checking propagates contexts and class tables top-down.



(b) Co-contextual type checking propagates context and class table requirements bottom-up.

Figure 8.3.: Contextual and co-contextual type checking.

the *size* requirements, we can already extract a conditional constraint that must hold if the requirements become mergeable, namely $(U_2 = U_4 \text{ if } U_1 = U_3)$. This constraint ensures the signatures of both *size* invocations are equal in case their receiver types U_1 and U_3 are equal. This way, we enable early error detection and incremental solving of constraints. Constraints can be solved continuously as soon as they have been generated in order to not wait for the whole program to be type checked. We discuss incremental type checking in more detail in Chapter 9.

After type checking the $+$ operator, the type checker encounters the class declarations of `List` and `LinkedList`. When type checking the class header `LinkedList extends List`, we have to record the inheritance relation between the two classes because methods can be invoked by `LinkedList`, but declared in `List`. For example, if `List` is not known to be a

superclass of `LinkedList` and given the declaration `List.add`, then we cannot just yet satisfy the requirement `LinkedList.add : Num \rightarrow U_3` . Therefore, we duplicate the requirement regarding `add` having as receiver `List`, i.e., `List.add : Num \rightarrow U_3` . By doing so, we can deduce the actual type of U_3 for the given declaration of `add` in `List`. Also, requirements regarding `size` are duplicated.

In the next step, the method declaration of `size` in `List` is type checked. Hence, we consider all requirements regarding `size`, i.e., $U_1.size : () \rightarrow U_2$ and $U_3.size : () \rightarrow U_4$. The receivers of `size` in both requirements are unknown. We cannot yet satisfy these requirements because we do not know whether U_1 and U_3 are equal to `List`, or not. To solve this, we introduce conditions as part of the requirements, to keep track of the relations between the unknown required classes and the declared ones. By doing so, we can deduce the actual types of U_2 and U_4 , and satisfy the requirements later, when we have more information about U_1 and U_3 .

Next, we encounter the method declaration `add` and satisfy the corresponding requirements. After satisfying the requirements regarding `add`, the type checker can infer the actual types of U_1 and U_3 . Therefore, we can also satisfy the requirements regarding `size`.

To summarize, during the co-contextual type checking of constructor calls, method invocations, and field accesses, the requirements flow bottom-up. Instead of looking up types of fields/methods in the class table, we introduce new class table requirements. These requirements are satisfied when the actual types of fields/methods become available.

9. Co-contextual Featherweight Java

In this chapter, we construct a co-contextual type checker for FJ. Initially, we describe the co-contextual structures for FJ, i.e., syntax, constraints, and class table requirements. Then, we provide a detailed description of the operations on the class table requirements. Next, we describe the co-contextual typing rules of FJ and how we apply dualism to build them from their contextual counterparts. We prove the equivalence between the contextual and co-contextual FJ and present the necessary theorems. Finally, we present the implementation of an efficient type checker for the co-contextual FJ. We illustrate optimizations applied to the implementation, such as, condition normalization, incremental and continuous constraint solving, and tree balancing. We evaluate the performance of the incremental type checker by comparing it to a contextual non-incremental FJ type checker, as well as to `javac` by translating Java programs that are limited to a subset of Java into FJ.

9.1. Co-Contextual Structures for Featherweight Java

In this section, we present the dual structure to class tables and operations on it to support the co-contextual formulation of FJ's type system. Specifically, we introduce bottom-up propagated *class table requirements*, replacing top-down propagated class tables.

9.1.1. Class Types and Constraints

For co-contextual FJ, we reuse the syntax of FJ in Chapter 8 (Figure 8.1), but extend the type language to *class types*:

U, V, \dots	Unification Variables
$T ::= C \mid U$	Class Types

We use constraints for refining class types, i.e., co-contextual FJ is a constraint-based type system. That is, next to class names, the type system may assign *unification variables*, designating unknowns in constraints. We further assume that there are countably many unification variables, equality of unification variables is decidable and that unification variables and class names are disjoint.

During bottom-up checking, we propagate sets S of constraints:

$s ::= T = T \mid T \neq T \mid T <: T \mid T \not<: T \mid T = T \text{ if } cond$	constraint
$S ::= \emptyset \mid S; s$	constraint set

A constraint s either states that two class types must be equal, non-equal, in a subtype

relation, non-subtype, or equal if some condition holds, which we leave underspecified for the moment.

9.1.2. Context Requirements

A typing context is a set of bindings from variables to types, while a context requirement is a set of bindings from variables to unification variables U . Operations on the typing context are lookup, extension, and duplication; their respective requirement context duals are: generating, removing, and merging. These operations are the same as the operations presented in co-contextual PCF in Chapter 3. Co-contextual FJ adopts context requirements and operations for method parameters and **this** unchanged.

9.1.3. Structure of Class Tables and Class Table Requirements

In the following, we describe the dual notion of a class table, called *class table requirements* and their operations. We first recapitulate the structure of FJ class tables [IK01], then stipulate the structure of class table requirements. Figure 9.1 shows the syntax of both. A class table is a collection of class definition clauses $CTcls$ defining the available classes.¹ A clause is a class name C followed by either the superclass, the signature of the constructor, a field type, or a method signature of C 's definition.

Contextual		Co-Contextual	
$CT ::= \emptyset$	class table	$CR ::= \emptyset$	class table req.
$CTcls \cup CT$		$(CReq, cond) \cup CR$	
$CTcls ::=$	def. clause	$CReq ::=$	class req.
$C \text{ extends } D$	extends clause	$T \text{ .extends: } T'$	inheritance req.
$C.\text{init}(\overline{C})$	ctor clause	$T.\text{init}(\overline{T})$	ctor req.
$C.f : C'$	field clause	$T.f : T'$	field req.
$C.m : \overline{C} \rightarrow C'$	method clause	$T.m : \overline{T} \rightarrow T'$	method req.
		$(T.m : \overline{T} \rightarrow T')_{opt}$	optional method req.
		$cond ::= \emptyset \mid T = T' \cup cond$	condition
		$T \neq T' \cup cond$	

Figure 9.1.: Class Table and Class Table Requirements Syntax.

As Figure 9.1 suggests, class tables and definition clauses in FJ have a counterpart in co-contextual FJ. Class tables become *class table requirements* CR , which are collections of pairs $(CReq, cond)$, where $CReq$ is a *class requirement* and $cond$ is its *condition*. Each class definition clause has a corresponding class requirement $CReq$, which is one of the following:

- A *inheritance requirement* $T \text{ .extends: } T'$, i.e., class type T must inherit from T' .

¹To make the correspondence to class table requirements more obvious, we show a decomposed form of class tables. The original FJ formulation [IK01] groups clauses by the containing class declaration.

- A *constructor requirement* $T.\text{init}(\bar{T}')$, i.e., class type T 's constructor signature must match \bar{T}' .
- A *field requirement* $T.f : T'$, i.e., class T (or one of its *supertypes*) must declare field f with class type T' .
- A *method requirement* $T.m : \bar{T}' \rightarrow T''$, i.e., class T (or one of its *supertypes*) must declare method m matching signature $\bar{T}' \rightarrow T''$.
- An *optional method requirement* $(T.m : \bar{T}' \rightarrow T'')_{opt}$, i.e., if the class type T declares the method m , then its signature must match $\bar{T}' \rightarrow T''$. While type checking method declarations, this requirement is used to ensure that method overrides in subclasses are well-defined. An optional method requirement is used as a counterpart of the conditional method lookup in rule T-METHOD of standard FJ, i.e., *if* $\text{mtype}(m, D, CT) = \bar{D} \rightarrow D_0$, *then* $\bar{C} = \bar{D}$; $C_0 = D_0$, where D is the superclass of the class C , in which the method declaration m under scrutiny is type checked, and \bar{C} , C_0 are the parameter and returned types of m as part of C .

A condition *cond* is a list of type equalities and inequalities, such that if one of them does not hold, the condition is unsatisfiable. Intuitively, $(CReq, cond)$ states that if the condition *cond* is unsatisfiable, then the requirement *CReq* is removed from the requirements set. With conditional requirements and constraints, we address the feature of nominal typing and inheritance for co-contextual FJ. In the following, we will describe their usage.

9.1.4. Operations on Class Tables and Requirements

In this section, we describe the co-contextual dual to FJ's class table operations as outlined in Figure 9.2.

We first consider FJ's lookup operations on class tables, which appear in premises of typing rules shown in Figure 8.2 to look up (1) fields, (2) field lists, (3) methods and (4) superclass lookup. The dual operation is to introduce a corresponding class requirement for the field, list of fields, method, or superclass.

Let us consider closely field lookup, i.e., $\text{field}(f_i, C, CT) = C_i$, meaning that class C in the class table CT has as member a field f_i of type C_i . We translate it to the dual operation of introducing a new class requirement $(C.f_i : U, \emptyset)$. Since we do not have any information about the type of the field, we choose a *fresh* unification variable U as type of field f_i . At the time of introducing a new requirement, its condition is empty.

Consider the next operation $\text{fields}(C, CT)$, which looks up all field members of a class. This lookup is used in the constructor call rule T-NEW; the intention is to retrieve the *constructor signature* in order to type check the subtyping relation between this signature and the types of expressions as parameters of the *constructor call*, i.e., $\bar{C} <: \bar{D}$ (rule T-NEW). As we can observe, the field names are not needed in this rule, only their types. Hence, in contrast to the original FJ rule [IK01], we deduce the constructor signature from fields lookup, rather than field names and their corresponding types, i.e.,

Contextual	Co-contextual
Field name lookup $\text{field}(f_i, C, CT) = C_i$	Class requirement for field $(C.f_i : U, \emptyset)$
Fields lookup $\text{fields}(C, CT) = C.\mathbf{init}(\bar{C})$	Class requirement for constructor $(C.\mathbf{init}(\bar{U}), \emptyset)$
Method lookup $\text{mtype}(m, C, CT) = \bar{C} \rightarrow C$	Class requirement for method $(C.m : \bar{U} \rightarrow U, \emptyset)$
Conditional method override $\text{if } \text{mtype}(m, C, CT) = \bar{C} \rightarrow C$	Optional class requirement for method $(C.m : \bar{U} \rightarrow U, \emptyset)_{opt}$
Super class lookup $\text{extends}(C, CT) = D$	Class requirement for super class $(C.\mathbf{extends} : U, \emptyset)$
Class table duplication $CT \rightarrow (CT, CT)$	Class requirement merging $\text{merge}_{CR}(CR_1, CR_2) = CR _S$ if all constraints in S hold

Figure 9.2.: Operations on class table and their co-contextual correspondence.

$\text{fields}(C, CT) = C.\mathbf{init}(\bar{D})$. The dual operation on class requirements is to add a new class requirement for the constructor, i.e., $(C.\mathbf{init}(\bar{U}), \emptyset)$. Analogously, the class table operations for method signature lookup and superclass lookup map to corresponding class table requirements.

Finally, standard FJ uses class table duplication to forward the class table to all parts of an FJ program, thus ensuring all parts are checked against the same context. The dual co-contextual operation, merge_{CR} , merges class table requirements originating from different parts of the program. Importantly, requirements merging needs to assure all parts of the program require compatible inheritance, constructors, fields, and methods for any given class. To merge two sets of requirements, we first identify the field and method names used in both sets and then compare the classes they belong to. The result of merging two sets of class table requirements CR_1 and CR_2 is a new set CR of class table requirements and a set of constraints, which ensure compatibility between the two original sets of overlapping requirements. Non-overlapping requirements get propagated unchanged to CR whereas potentially overlapping requirements receive special treatment depending on the requirement kind.

The full merge definition is shown in Appendix B.1; Figure 9.3 shows the merge operation for overlapping method requirements, which results in a new set of requirements CR_m and constraints S_m . To compute CR_m , we identify method requirements on the equally-named methods m in both sets and distinguish two cases. First, if the receivers are different $T_1 \neq T_2$, then the requirements are not actually overlapping. We retain the two requirements unchanged, except that we remember the failed condition for future reference. Second, if the receivers are equal $T_1 = T_2$, then the requirements are actually overlapping. We merge them into a single requirement and produce corresponding constraints in S_m . One of the key benefits of keeping track of conditions in class table requirements is that often these conditions allow us to discharge requirements early on when their conditions

$$\begin{aligned}
 CR_m = & \{(T_1.m : \overline{T_1} \rightarrow T'_1, cond_1 \cup (T_1 \neq T_2)) \\
 & \cup (T_2.m : \overline{T_2} \rightarrow T'_2, cond_2 \cup (T_1 \neq T_2)) \\
 & \cup (T_1.m : \overline{T_1} \rightarrow T'_1, cond_1 \cup cond_2 \cup (T_1 = T_2)) \\
 & \mid (T_1.m : \overline{T_1} \rightarrow T'_1, cond_1) \in CR_1 \wedge (T_2.m : \overline{T_2} \rightarrow T'_2, cond_2) \in CR_2\} \\
 S_m = & \{(T'_1 = T'_2 \text{ if } T_1 = T_2) \cup (\overline{T_1} = \overline{T_2} \text{ if } T_1 = T_2) \\
 & \mid (T_1.m : \overline{T_1} \rightarrow T'_1, cond_1) \in CR_1 \wedge (T_2.m : \overline{T_2} \rightarrow T'_2, cond_2) \in CR_2\}
 \end{aligned}$$

 Figure 9.3.: Merge operation of method requirements CR_1 and CR_2 .

are unsatisfiable. In particular, in Section 9.4 we describe a compact representation of conditional requirements that facilitates early pruning and is paramount for good performance. However, the main reason for conditional class table requirements is their removal, which we discuss subsequently.

9.1.5. Class Table Construction and Requirements Removal

Our formulation of the contextual FJ type system differs in the presentation of the class table compared to the original paper [IK01]. Whereas Igarashi et al. assume that the class table is a pre-defined static structure, we explicitly consider its formation through a sequence of operations. The class table is initially empty and gradually extended with class table clauses $CTcls$ for each class declaration L of a program. Dual to that, class table requirements are initially unsatisfied and gradually removed. We define an operation for *adding* clauses to the class table and a corresponding co-contextual dual operation on class table requirements for *removing* requirements.

Figure 9.4 shows a collection of adding and removing operations for every possible kind of class table clause $CTcls$.

Contextual	Co-contextual
Class table empty $CT = \emptyset$	Unsatisfied class requirements CR
Adding extend $\text{addExt}(L, CT)$	Remove extend $\text{removeExt}(L, CR)$
Adding constructor $\text{addCtor}(L, CT)$	Remove constructor $\text{removeCtor}(L, CR)$
Adding fields $\text{addFs}(L, CT)$	Remove fields $\text{removeFs}(L, CR)$
Adding methods $\text{addMs}(L, CT)$	Remove methods $\text{removeMs}(L, CR)$

Figure 9.4.: Constructing class table and their co-contextual correspondence.

In general, clauses are added to the class table starting from superclass to subclass declarations. For a given class, the class header with **extends** is added before the other clauses. Dually, we start removing requirements that correspond to clauses of a subclass,

followed by those corresponding to clauses of superclass declarations. For a given class, we first remove requirements corresponding to method, fields, or constructor clauses, then those corresponding to the class header **extends** clause. Note that our sequencing still allows for mutual class dependencies. For example, the following is a valid sequence of clauses where A depends on B and vice versa:

class A **extends** Object; **class** B **extends** Object; A.m: () → B; B.m: () → A.

The full definition of the addition and removal operations for all cases of clause definition is shown in Appendix B.1; Figure 9.5 presents the definitions of adding and removing method and **extends** clauses.

Remove operations for method clauses. The function `removeMs` removes a list of methods by applying the function `removeM` to each of them. `removeM` removes a single method declaration defined in class C . To this end, `removeM` identifies requirements on the same method name m and refines their receiver to be different from the removed declaration’s defining class. That is, the refined requirement $(T.m : \dots, \text{cond} \cup (T \neq C))$ only requires method m if the receiver T is different from the defining class C . If the receiver T is, in fact, equal to C , then the condition of the refined requirement is unsatisfiable and the requirement can be discharged. To ensure the required type also matches the declared type, `removeM` also generates conditional constraints in the case $T = C$. Note that whether $T = C$ can often not be determined immediately because T may be a placeholder type U .

We illustrate the removal of methods using the class declaration of `List` shown in Section 8.2. Consider the class requirement set $CR = \{(U_1.\text{size}() \rightarrow U_2, \emptyset)\}$. Encountering the declaration of method `add` has no effect on this set because there is no requirement on `add`. However, when encountering the declaration of method `size`, we refine the set as follows:

$$\text{removeM}(\text{List}, \text{Int } \text{size}() \{ \dots \}, CR) = \{(U_1.\text{size} : () \rightarrow U_2, U_1 \neq \text{List})\}_S$$

with a new constraint $S = \{U_2 = \text{Int} \text{ if } U_1 = \text{List}\}$. Thus, we have satisfied the requirement in CR for $U_1 = \text{List}$ because we know the actual return type of `size`, which is `Int`, and the condition $U_1 = \text{List}$ in the constraint S is satisfiable. We only leave the requirement in case U_1 represents another type. In particular, if we learn at some point that U_1 indeed represents `List`, we can discharge the requirement because its condition is unsatisfiable. This is important because a program is only closed and well-typed if its requirement set is empty.

Remove operations for extends clauses. The function `removeExt` removes the **extends** clauses (C . **extends** D). This function, in addition to identifying the requirements regarding **extends** and following the same steps as above for `removeM`, duplicates all requirements for fields and methods. The duplicate introduces a requirement the same as the existing one, but with a different receiver, which is the superclass D that potentially declares the required fields and methods. The conditions also change. We add to the existing requirements cond a type inequality $(T \neq C)$, to encounter the case when the receiver T is actually replaced by C , but it is required to have a certain field

$$\text{addMs}(C, \overline{M}, CT) = \overline{C.m : \overline{C} \rightarrow C'} \cup CT$$

$$\text{removeM}(C, C' \ m(\overline{C} \ \overline{x}) \ \{\mathbf{return} \ e\}, CR) = CR'|_S$$

$$\begin{aligned} \text{where } CR' = & \{(T.m : \overline{T} \rightarrow T', \text{cond} \cup (T \neq C)) \mid (T.m : \overline{T} \rightarrow T', \text{cond}) \in CR\} \\ & \cup (CR \setminus \overline{(T.m : \overline{T} \rightarrow T', \text{cond})}) \end{aligned}$$

$$S = \{(T' = C' \text{ if } T = C) \cup (\overline{T} = \overline{C} \text{ if } T = C) \mid (T.m : \overline{T} \rightarrow T', \text{cond}) \in CR\}$$

$$\text{removeMs}(C, \overline{M}, CR) = CR'|_S$$

$$\text{where } CR' = \{CR_m \mid (C' \ m(\overline{C} \ \overline{x}) \ \{\mathbf{return} \ e\}) \in \overline{M}\}$$

$$\wedge \text{removeM}(C, C' \ m(\overline{C} \ \overline{x}) \ \{\mathbf{return} \ e\}, CR) = CR_m|_{S_m}\}$$

$$S = \{S_m \mid (C' \ m(\overline{C} \ \overline{x}) \ \{\mathbf{return} \ e\}) \in \overline{M}\}$$

$$\wedge \text{removeM}(C, C' \ m(\overline{C} \ \overline{x}) \ \{\mathbf{return} \ e\}, CR) = CR_m|_{S_m}\}$$

$$\text{addExt}(\mathbf{class} \ C \ \mathbf{extends} \ D, CT) = (C \ \mathbf{extends} \ D) \cup CT$$

$$\text{removeExt}(\mathbf{class} \ C \ \mathbf{extends} \ D, CR) = CR'|_S$$

$$\text{where } CR' = \{(T.\mathbf{extends} : T', \text{cond} \cup (T \neq C)) \mid (T.\mathbf{extends} : T', \text{cond}) \in CR\}$$

$$\cup \{(T.m : \overline{T} \rightarrow T', \text{cond} \cup (T \neq C))\}$$

$$\cup (D.m : \overline{T} \rightarrow T', \text{cond} \cup (T = C)) \mid (T.m : \overline{T} \rightarrow T', \text{cond}) \in CR\}$$

$$\cup \{(T.m : \overline{T} \rightarrow T', \text{cond} \cup (T \neq C))_{opt}\}$$

$$\cup (D.m : \overline{T} \rightarrow T', \text{cond} \cup (T = C))_{opt}$$

$$\mid (T.m : \overline{T} \rightarrow T', \text{cond})_{opt} \in CR\}$$

$$\cup \{(T.f : T', \text{cond} \cup (T \neq C)) \cup (D.f : T', \text{cond} \cup (T = C))\}$$

$$\mid (T.f : T', \text{cond}) \in CR\}$$

$$S = \{(T' = D \text{ if } T = C) \mid (T.\mathbf{extends} : T', \text{cond}) \in CR\}$$

Figure 9.5.: Add and remove operations of method and extends clauses.

or method, which is declared in D , the superclass of T . This requirement should be discharged because we know the actual type of the required field or method, which is inherited from the given declaration in D . Also, we add a type equality to the duplicate requirement $T = C$, because this requirement will be discharged when we encounter the actual declarations of fields or methods in the superclass.

We illustrate the removal of **extends** using the class declaration **LinkedList extends List**. Consider the requirement set $CR = \{(U_3.size : () \rightarrow U_4, \emptyset)\}$. We encounter the

9. Co-contextual Featherweight Java

declaration for `LinkedList` and the requirement set changes as follows:

$$\text{removeExt}(\text{class LinkedList extends List}, CR) = \{(U_3.\text{size} : () \rightarrow U_4, U_3 \neq \text{LinkedList}), (\text{List}.\text{size} : () \rightarrow U_4, U_3 = \text{LinkedList})\}_{|S},$$

where $S = \emptyset$. S is empty, because there are no requirements on **extends**. If we learn at some point that $U_3 = \text{LinkedList}$, then the requirement $(U_3.\text{size} : () \rightarrow U_4, U_3 \neq \text{LinkedList})$ is discharged because its condition is unsatisfiable. Also, if we learn that size is declared in `List`, then $(\text{List}.\text{size} : () \rightarrow U_4, U_3 = \text{LinkedList})$ is discharged applying `removeM`, as shown above, and U_4 can be replaced by its actual type.

Usage and necessity of conditions. As shown throughout this section, conditions play an important role to enable merging and removal of requirements over nominal receiver types and to support inheritance. Because of nominal typing, field and method lookup depends on the name of the defining context and we do not know the actual type of the receiver class when encountering a field or method reference. Thus, it is impossible to deduce their types until more information is known. Moreover, if a class is required to have fields/methods, which are actually declared in a superclass of the required class, then we need to deduce their actual type/signature and meanwhile fulfill the respective requirements. For example, considering the requirement $U_3.\text{size} : () \rightarrow U_4$, if $U_3 = \text{LinkedList}$, `LinkedList extends List`, and size is declared in `List`, then we have to deduce the actual type of U_4 and satisfy this requirement. To overcome these obstacles we need additional structure to maintain the relations between the required classes and the declared ones, and also to reason about the partial fulfillment of requirements. Conditions come to place as the needed structure to maintain these relations and indicate the fulfillment of requirements.

9.2. Co-Contextual Featherweight Java Typing Rules

In this section, we derive co-contextual FJ's typing rules systematically from FJ's typing rules. The main idea is to transform the rules into a form that eliminates any context dependencies that require top-down propagation of information.

Concretely, context and class table requirements (Section 9.1) in output positions to the right replace typing contexts and class tables in input positions to the left. Additionally, co-contextual FJ propagates constraint sets S in output positions. Note that the program typing judgment does not change, because programs are closed, requiring neither typing context nor class table inputs. Correspondingly, neither context nor class table requirements need to be propagated as outputs.

Figure 9.6 shows the co-contextual FJ typing rules (the reader may want to compare against contextual FJ in Chapter 8 in Figure 8.2). In what follows, we will discuss the rules for each kind of judgment.

TC-VAR	$\frac{U \text{ is fresh}}{x : U \mid \emptyset \mid x : U \mid \emptyset}$
TC-FIELD	$\frac{e : T_e \mid S_e \mid R_e \mid CR_e \quad CR _{S_f} = \text{merge}_{CR}(CR_e, (T_e.f_i : U, \emptyset)) \quad U \text{ is fresh}}{e.f_i : U \mid S_e \cup S_f \mid R_e \mid CR}$
TC-INVK	$\frac{e : T_e \mid S_e \mid R_e \mid CR_e \quad \overline{e : T \mid S \mid R \mid CR} \quad CR_m = (T_e.m : \overline{U} \rightarrow U', \emptyset) \quad \overline{S_s} = \{T <: U\} \quad U', \overline{U} \text{ are fresh} \quad R' _{S_r} = \text{merge}_R(R_e, \overline{R}) \quad CR' _{S_{cr}} = \text{merge}_{CR}(CR_e, CR_m, \overline{CR})}{e.m(\overline{e}) : U' \mid \overline{S} \cup S_e \cup \overline{S_s} \cup S_r \cup S_{cr} \mid R' \mid CR'}$
TC-NEW	$\frac{\overline{e : T \mid S \mid R \mid CR} \quad CR_f = (C.\text{init}(\overline{U}), \emptyset) \quad \overline{S_s} = \{T <: \overline{U}\} \quad \overline{U} \text{ is fresh} \quad R' _{S_r} = \text{merge}_R(\overline{R}) \quad CR' _{S_{cr}} = \text{merge}_{CR}(CR_f, \overline{CR})}{\text{new } C(\overline{e}) : C \mid \overline{S} \cup \overline{S_s} \cup S_r \cup S_{cr} \mid R' \mid CR'}$
TC-UCAST	$\frac{e : T_e \mid S_e \mid R_e \mid CR_e \quad S_s = \{T_e <: C\}}{(C)e : C \mid S_e \cup S_s \mid R_e \mid CR_e}$
TC-DCAST	$\frac{e : T_e \mid S_e \mid R_e \mid CR_e \quad S_s = \{C <: T_e\} \quad S_n = \{C \neq T_e\}}{(C)e : C \mid S_e \cup S_s \cup S_n \mid R_e \mid CR_e}$
TC-SCAST	$\frac{e : T_e \mid S_e \mid R_e \mid CR_e \quad S_s = \{C \not<: T_e\} \quad S'_s = \{T_e \not<: C\}}{(C)e : C \mid S_e \cup S_s \cup S'_s \mid R_e \mid CR_e}$
TC-METHOD	$\frac{e : T_e \mid S_e \mid R_e \mid CR_e \quad \overline{S_x} = \{C = R_e(x) \mid x \in \text{dom}(R_e)\} \quad S_c = \{U_c = R_e(\text{this}) \mid \text{this} \in \text{dom}(R_e)\} \quad S_s = \{T_e <: C_0\} \quad R_e - \text{this} - \overline{x} = \emptyset \quad U_c, U_d \text{ are fresh} \quad CR _{S_{cr}} = \text{merge}_{CR}(CR_e, (U_c.\text{extends}: U_d, \emptyset), (U_d.m : \overline{C} \rightarrow C_0, \emptyset)_{opt})}{C_0 \ m(\overline{C} \ \overline{x}) \ \{\text{return } e\} \ \text{OK} \mid S_e \cup S_s \cup S_c \cup S_{cr} \cup \overline{S_x} \mid U_c \mid CR}$
TC-CLASS	$\frac{K = C(\overline{D}' \ \overline{g}, \overline{C}' \ \overline{f})\{\text{super}(\overline{g}); \text{this}.\overline{f} = \overline{f}\} \quad \overline{M} \text{ OK} \mid S \mid U \mid \overline{CR} \quad CR' _{S_{cr}} = \text{merge}_{CR}((D.\text{init}(\overline{D}'), \emptyset), \overline{CR}) \quad \overline{S_{eq}} = \{U = C\}}{\text{class } C \text{ extends } D\{\overline{C}' \ \overline{f}; K \ \overline{M}\} \text{ OK} \mid \overline{S} \cup \overline{S_{eq}} \cup S_{cr} \mid CR'}$
TC-PROGRAM	$\frac{\overline{L} \text{ OK} \mid S \mid \overline{CR} \quad \text{merge}_{CR}(\overline{CR}) = CR' _{S'} \quad \uplus_{L' \in \overline{L}} (\text{removeMs}(CR', L') \uplus \text{removeFs}(CR', L')) \uplus \text{removeCtor}(CR', L') \uplus \text{removeExt}(CR', L') = \emptyset _S}{\overline{L} \text{ OK} \mid \overline{S} \cup S' \cup S}$

Figure 9.6.: A co-contextual formulation of the type system of Featherweight Java.

9.2.1. Expression Typing

The typing rule TC-VAR is dual to the standard variable lookup rule T-VAR . It marks a distinct *occurrence* of x (or the self reference **this**) by assigning a fresh unification variable U . Furthermore, it introduces a new context requirement $\{x : U\}$, as the dual operation of context lookup for variables x ($\Gamma(x) = C$) in T-VAR . Since the latter does not access the class table, dually, TC-VAR outputs empty class table requirements.

The typing rule TC-FIELD is dual to T-FIELD for field accesses. The latter requires a field name lookup (field), which, dually, translates to a new class requirement for the field f_i , i.e., $(T_e.f_i : U, \emptyset)$ (cf. Section 9.1). Here, T_e is the class type of the receiver e . U is a fresh unification variable, marking a distinct occurrence of field name f_i , which is the class type of the entire expression. Furthermore, we merge the new field requirement with the class table requirements CR_e propagated from e . The result of merging is a new set of requirements CR and a new set of constraints S_{cr} . Just as the context Γ is passed into the subexpression e in T-FIELD , we propagate the context requirements for e for the entire expression. Finally, we propagate both the constraints S_e for e and the merge constraints S_f as the resulting output constraints.

The typing rule TC-INVK is dual to T-INVK for method invocations. Similarly to field access, the dual of method lookup is introducing a requirement for the method m and merge it with the requirements from the premises. Again, we choose fresh unification variables for the method signature $\bar{U} \rightarrow U'$, marking a distinct occurrence of m . We type check the list \bar{e} of parameters, adding a subtype constraint $\bar{T} <: \bar{U}$, corresponding to the subtype check in T-INVK . Finally, we merge all context and class table requirements propagated from the receiver e and the parameters \bar{e} , and all the constraints.

The typing rule TC-NEW is dual to T-NEW for object creation. We add a new class requirement $C.\text{init}(\bar{U})$ for the constructor of class C , corresponding to the *fields* operation in FJ. We cannot look up the fields of C in the class table, therefore we assign fresh unification variables \bar{U} for the constructor signature. We add the subtyping constraint $\bar{T} <: \bar{U}$ for the parameters, analogous to the subtype check in T-NEW . As in the other rules, we propagate a collective merge of the propagated requirement structures/constraints from the subexpressions with the newly created requirements/constraints.

Typing rules for casts, i.e., TC-UCAST , TC-DCAST and TC-SCAST are straightforward adaptations of their contextual counterparts following the same principles. These three type rules do overlap. We do not distinguish them in the formalization, but to have an algorithmic formulation, we implement different node names for each of them. That is, typing rules for casts are syntactically distinguished.

9.2.2. Method Typing

The typing rule TC-METHOD is dual to T-METHOD for checking method declarations. For checking the method body, the contextual version extends the empty typing context with entries for the method parameters \bar{x} and the self-reference **this**, which is implicitly in scope. Dually, we remove the requirements on the parameters and self-reference in R_e propagated from the method body. Corresponding to extending an empty context, the

removal should leave no remaining requirements on the method body. Furthermore, the equality constraints $\overline{S_x}$ ensure that the annotated class types for the parameters agree with the class types in R_e .² This corresponds to binding the parameters to the annotated classes in a typing context. Analogously, the constraints S_c deal with the self-reference. For the latter, we need to know the associated class type, which in the absence of the class table is at this point unknown. Hence, we assign a fresh unification variable U_c for the yet to be determined class containing the method declaration. The contextual rule T-METHOD further checks if the method declaration correctly overrides another method declaration in the superclass, that is, if it exists in the superclass must have the same type. We choose another fresh unification variable U_d for the yet to be determined superclass of U_c and add appropriate supertype and optional method override requirements. We assign to the optional method requirement $U_d.m$ the type of m declared in U_c . If later is known that there exists a declaration for m in the actual type of U_d , the optional requirement is considered and equality constraints are generated. These constraints ensure that the required type of m in the optional requirement is the same as the provided type of m in the actual superclass of U_c . Otherwise this optional method requirement is invalidated and not considered. By doing so, we enable the feature of subtype polymorphism for co-contextual FJ. Finally, we add the subtype constraint ensuring that the method body's type is conforming to the annotated return type.

9.2.3. Class Typing

The typing rule TC-CLASS is used for checking class declarations. A declaration of a given class C provides definite information on the identity of unification variable U for the enclosing class type, because we type check each method declaration independently. Therefore, we add the constraints $\{U = C\}$, effectively completing the method declarations with their enclosing class C .

9.2.4. Program Typing

The typing rule TC-PROGRAM checks a list of class declarations \overline{L} . Class declarations of all classes provide a definite information on the identity of their superclasses, constructor, fields, methods signatures. Dual to adding clauses in the class table by constructing it, we remove requirements with respect to the provided information from the declarations. Hence, dually to the class table being fully extended with clauses from all class declarations, requirements set is empty. The result of removing different clauses is a new set of requirements and a set of constraints. Hence, we use notation \uplus to express the union of the returned tuples (requirements and constraints), i.e., $CR|_S \uplus CR'|_{S'} = CR \cup CR'|_{S \cup S'}$.

As shown, we can systematically derive co-contextual typing rules for Featherweight Java through duality.

²Note that a parameter x occurs in the method body if and only if there is a requirement for x in R_e (i.e., $x \in \text{dom}(R_e)$), which is due to the bottom-up propagation. The same holds for the self-reference **this**.

9.3. Typing Equivalence

In this section, we prove the typing equivalence of expressions, methods, classes, and programs between FJ and co-contextual FJ. That is, (1) we want to convey that an expression, method, class and program is type checked in FJ if and only if it is type checked in co-contextual FJ, and (2) that there is a correspondence relation between typing constructs for each typing judgment.

We use σ to represent substitution, which is a set of bindings from unification variables to class types ($\{U \mapsto C\}$). $\text{projExt}(CT)$ is a function that given a class table CT returns the immediate subclass relation Σ of classes in CT . That is, $\Sigma := \{(C_1, C_2) \mid (C_1 \text{ extends } C_2) \in CT\}$. Given a set of constraints S and a relation between class types Σ , where $\text{projExt}(CT) = \Sigma$, then the solution to that set of constraints is a substitution, i.e., $\text{solve}(S, \Sigma) = \sigma$. Also we assume that every element of the *class table*, i.e., supertypes, constructors, fields and methods types are class type, namely ground types. Ground types are types that cannot be substituted.

Initially, we prove equivalence for expressions. We describe the *correspondence relation*, which states that a) the types of expressions are the same in both formulations, b) provided variables in context are more than required ones in context requirements and c) provided class members are more than required ones. Intuitively, an expression to be well-typed in co-contextual FJ should have all requirements satisfied. Context requirements are satisfied when for all required variables, we find the corresponding bindings in context. Class table requirements are satisfied, when for all valid requirements we can find a corresponding declaration in a class of the same type as the required one, or in its superclasses. A requirement is valid, when its condition is satisfiable, i.e., all type equalities and inequalities in the condition hold. The relation between class table and class requirements is formally defined in the Appendix B.2.

Definition 1 (Correspondence relation for expressions). *Given judgments $\Gamma; CT \vdash e : C$, $e : T \mid S \mid R \mid CR$, and $\text{solve}(\Sigma, S) = \sigma$, where $\text{projExt}(CT) = \Sigma$. The correspondence relation between Γ and R , CT and CR , written $(C, \Gamma, CT) \triangleright \sigma(T, R, CR)$, is defined as:*

- a) $C = \sigma(T)$
- b) $\Gamma \supseteq \sigma(R)$
- c) CT satisfies $\sigma(CR)$

We stipulate two different theorems to state both directions of equivalence for expressions.

Theorem 2 (Equivalence of expressions: \Rightarrow). *Given e , C , Γ , CT , if $\Gamma; CT \vdash e : C$, then there exists T , S , R , CR , Σ , σ , where $\text{projExt}(CT) = \Sigma$ and $\text{solve}(\Sigma, S) = \sigma$, such that $e : T \mid S \mid R \mid CR$ holds, σ is a ground solution and $(C, \Gamma, CT) \triangleright \sigma(T, R, CR)$ holds.*

Theorem 3 (Equivalence of expressions: \Leftarrow). *Given e , T , S , R , CR , Σ , if $e : T \mid S \mid R \mid CR$, $\text{solve}(\Sigma, S) = \sigma$, and σ is a ground solution, then there exists C , Γ , CT , such that $\Gamma; CT \vdash e : C$, $(C, \Gamma, CT) \triangleright \sigma(T, R, CR)$ and $\text{projExt}(CT) = \Sigma$.*

Theorems 2 and 3 are proved by induction on the typing judgment of expressions. The most challenging aspect consists in proving the relation between the class table and class table requirements. In Theorem 2, the class table is given and the requirements are a collective merge of the propagated requirement from the subexpressions with the newly created requirements. In Theorem 3, the class table is not given, therefore we construct it through the information retrieved from *ground class requirements*. We ensure class table correctness and completeness with respect to the given requirements. First, we ensure that the class table we construct is correct, i.e., types of **extends**, fields, and methods clauses we add in the class table are equal to the types of the same **extends**, fields, and methods if they already exist in the class table. Second, we ensure that the class table we construct is complete, i.e., the constructed class table satisfies all given requirements.

Next, we present the theorem of equivalence for methods. The difference from expressions is that there is no context, therefore no relation between context and context requirements is required. Instead, the fresh unification variable introduced in co-contextual FJ as a placeholder for the actual class, where the method under scrutiny is type checked in, after substitution should be the same as the class where the method is type checked in FJ.

Theorem 4 (Equivalence of methods: \Rightarrow). *Given m, C, CT , if $C; CT \vdash C_0 m(\overline{C} \ \overline{x}) \{\text{return } e\} \text{ OK}$, then there exists S, T, CR, Σ, σ , where $\text{projExt}(CT) = \Sigma$ and $\text{solve}(\Sigma, S) = \sigma$, such that $C_0 m(\overline{C} \ \overline{x}) \{\text{return } e_0\} \text{ OK} \mid S \mid T \mid CR$ holds, σ is a ground solution and $(C, CT) \triangleright_m \sigma(T, CR)$ holds.*

Theorem 5 (Equivalence of methods: \Leftarrow). *Given m, T, S, CR, Σ , if $C_0 m(\overline{C} \ \overline{x}) \{\text{return } e_0\} \text{ OK} \mid S \mid T \mid CR$, $\text{solve}(\Sigma, S) = \sigma$, and σ is a ground solution, then there exists C, CT , such that $C; CT \vdash C_0 m(\overline{C} \ \overline{x}) \{\text{return } e\} \text{ OK}$ holds, $(C, CT) \triangleright_m \sigma(T, CR)$ and $\text{projExt}(CT) = \Sigma$.*

Theorems 5 and 6 are proved by induction on the typing judgment. The difficulty increases in proving equivalence for methods because we have to consider the optional requirement, as introduced in the previous sections. To prove the relation between the class table and optional requirements is required a different strategy; we accomplish the proof by using case distinction. We have a detailed proof for method declaration, and also how this affects class table construction, and we prove a correct and complete construction of it.

Lastly, we present the theorem of equivalence for classes and programs.

Theorem 6 (Equivalence of classes: \Rightarrow). *Given C, CT , if $CT \vdash \text{class } C \text{ extends } D \{\overline{C} \ \overline{f}; K \ \overline{M}\} \text{ OK}$, then there exists S, CR, Σ, σ , where $\text{projExt}(CT) = \Sigma$ and $\text{solve}(\Sigma, S) = \sigma$, such that $\text{class } C \text{ extends } D \{\overline{C} \ \overline{f}; K \ \overline{M}\} \text{ OK} \mid S \mid CR$ holds, σ is a ground solution and $(CT) \triangleright_c \sigma(CR)$ holds.*

Theorem 7 (Equivalence of classes: \Leftarrow). *Given C, CR, Σ , if $\text{class } C \text{ extends } D \{\overline{C} \ \overline{f}; K \ \overline{M}\} \text{ OK} \mid S \mid CR$, $\text{solve}(\Sigma, S) = \sigma$, and σ is a ground solution, then there exists CT , such that*

9. Co-contextual Featherweight Java

$CT \vdash \text{class } C \text{ extends } D \{ \overline{C} \ \overline{f}; K \ \overline{M} \} \text{ OK holds, } (CT) \triangleright_c \sigma(CR) \text{ holds and } \text{projExt}(CT) = \Sigma.$

Theorems 8 and 9 are proved by induction on the typing judgment. Class declaration requires to prove only the relation between the class table and class table requirements since there is no context.

Typing rule for programs does not have as inputs context and class table, therefore there is no relation between context, class table and requirements. The equivalence theorem describes that a program in FJ and co-contextual FJ is well-typed.

Theorem 8 (Equivalence for programs: \Rightarrow). *Given \overline{L} , if $\overline{L} \text{ OK}$, then there exists S, Σ, σ , where $\text{projExt}(\overline{L}) = \Sigma$ and $\text{solve}(\Sigma, S) = \sigma$, such that $\overline{L} \text{ OK} \mid S$ holds and σ ground solution.*

Theorem 9 (Equivalence for programs: \Leftarrow). *Given \overline{L} , if $\overline{L} \text{ OK} \mid S$, $\text{solve}(\Sigma, S) = \sigma$, where $\text{projExt}(\overline{L}) = \Sigma$, and σ is a ground solution, then $\overline{L} \text{ OK}$ holds.*

Theorems 10 and 11 are proved by induction on the typing judgment. In here, we prove that a class table containing all clauses provided from the given class declarations is dual to empty class table requirements in the inductive step.

Omitted definitions, lemmas and proofs can be found in Appendix B.2.

9.4. Efficient Incremental FJ Type Checking

The co-contextual FJ model from Section 9.1 and 9.2 was designed such that it closely resembles the formulation of the original FJ type system, where all differences are motivated by dually replacing contextual operations with co-contextual ones. As such, this model served as a good basis for the equivalence proof from the previous section. However, to obtain a type checker implementation for co-contextual FJ that is amenable to efficient incrementalization, we have to employ a number of behavior-preserving optimizations. In the present section, we describe these optimization and the resulting *incremental* type checker implementation for co-contextual FJ. The source code is available online at

<https://github.com/seba--/incremental>.

Condition normalization. In our formal model from Section 9.1 and 9.2, we represent context requirements as a set of conditional class requirements $CR \subset Creq \times cond$. Throughout type checking, we add new class requirements using function merge, but we only discharge class requirements in rule TC-PROGRAM at the very end of type checking. Since merge generates $3 * m * n$ conditional requirements for inputs with m and n requirements respectively, requirements quickly become intractable even for small programs.

The first optimization we conduct is to eagerly normalize conditions of class requirements. Instead of representing conditions as a type of type equalities and inequalities, we map receiver types to the following condition representation (shown as Scala code):

```
case class Condition(notGround: Set[CName], notVar: Set[UCName],
                    sameVar: Set[UCName], sameGroundAlternatives: Set[CName]).
```


A condition is true if the receiver type is different from all ground types (CName) and unification variables (UCName) in `notGround` and `notVar`, if the receiver type is equal to all unification variables in `sameVar`, and if `sameGroundAlternatives` is either empty or the receiver type occurs in it. That is, if `sameGroundAlternatives` is non-empty, then it stores a set of alternative ground types, one of which the receiver type must be equal to.

When adding an equation or inequation to the condition over a receiver type, we check whether the condition becomes unsatisfiable. For example, when equating the receiver type to the ground type `C` and `notGround.contains(C)`, we mark the resulting condition to be unsatisfiable. Recognizing unsatisfiable conditions has the immediate benefit of allowing us to discard the corresponding class requirements right away. Unsatisfiable conditions occur quite frequently because `merge` generates both equations and inequations for all receiver types that occur in the two merged requirement sets.

If a condition is not unsatisfiable, we normalize it such that the following assertions are satisfied:

- (i) the receiver type does not occur in any of the sets
- (ii) `sameGroundAlternatives.isEmpty` || `notGround.isEmpty`
- (iii) `notVar.intersect(sameVar).isEmpty`.

Since normalized conditions are more compact, this optimization saves memory and time required for memory management. Moreover, it makes it easy to identify irrefutable conditions, which is the case exactly when all four sets are empty, meaning that there are no further requisites on the receiver type. Such knowledge is useful when `merge` generates conditional constraints, because irrefutable conditions can be ignored. Finally, condition normalization is a prerequisite for the subsequent optimization.

In-depth merging of conditional class requirements. In PCF (Chapter3), the number of requirements of an expression was bound by the number of free variables that occur in that expression. To this end, the `merge` operation used for co-contextual PCF identifies subexpression requirements on the same free variable and merges them into a single requirement. For example, the expression $x + x$ has only one requirement $\{x : U_1\} \mid \{U_1 = U_2\}$, even though the two subexpressions propagate two requirements $\{x : U_1\}$ and $\{x : U_2\}$, respectively.

Unfortunately, the `merge` operation of co-contextual FJ given in Section 9.1.2 does not enjoy this property. Instead of merging requirements, it merely collects them and updates their conditions. A more in-depth merge of requirements is possible whenever two code fragments require the same member from the same receiver type. For example, the expression `this.x + this.x` needs only one requirement $\{U_1.x() : U_2\} \mid \{U_1 = U_3, U_2 = U_4\}$, even though the two subexpressions propagate two requirements $\{U_1.x() : U_2\}$ and $\{U_3.x() : U_4\}$, respectively. Note that $U_1 = U_3$ because of the use of `this` in both subexpressions, but $U_2 = U_4$ because of the in-depth merge.

However, conditions complicate the in-depth merging of class requirements: We may only merge two requirements if we can also merge their conditions. That is, for conditional requirements $(creq_1, cond_1)$ and $(creq_2, cond_2)$ with the same receiver type, the merged requirement must have the condition $cond_1 \vee cond_2$. In general, we cannot express $cond_1 \vee cond_2$ using our Condition representation from above because all fields except

`sameGroundAlternatives` represent conjunctive prerequisites, whereas `sameGroundAlternatives` represents disjunctive prerequisites. Therefore, we only support in-depth merging when the conditions are identical up to `sameGroundAlternatives` and we use the union operator to combine their `sameGroundAlternatives` fields.

This optimization may seem a bit overly specific to certain use cases, but it turns out it is generally applicable. The reason is that function `removeExt` creates requirements of the form $(D.f : T', \text{cond} \cup (T = C_i))$ transitively for all subclasses C_i of D where no class between C_i and D defines field f . Our optimization combines these requirements into a single one, roughly of the form $(D.f : T', \text{cond} \cup (T = \bigvee_i C_i))$. Basically, this requirement concisely states that D must provide a field f of type T' if the original receiver type T corresponds to any of the subclasses C_i of D .

Incrementalization and continuous constraint solving. We adopt the general incrementalization strategy from co-contextual PCF (Chapter 3): Initially, type check the full program bottom-up and memoize the typing output for each node (including class requirements and constraint system). Then, upon a change to the program, recheck each node from the change to the root of the program, reusing the memoized results from unchanged subtrees. This way, incremental type checking asymptotically requires only $\log n$ steps for a program with n nodes.

In our formal model of co-contextual FJ, we collect constraints during type checking and solve them at the end to yield a substitution for the unification variables. As discussed in Chapter 3 for co-contextual PCF, this strategy is inadequate for incremental type checking, because we would memoize unsolved constraints and thus only obtain an incremental constraint generator, but even a small change would entail that all constraints had to be solved from scratch.

In our implementation, we follow PCF’s strategy of continuously solving constraints as soon as they are generated, memoizing the resulting partial constraint solutions. In particular, equality constraints that result from merge and remove operations can be solved immediately to yield a substitution, while subtype constraints often have to be deferred until more information about the inheritance hierarchy is available. In the context of FJ with its nominal types, continuous constraint solving has the added benefit of enabling additional requirement merging, for example, because two method requirements share the same receiver type after substitution.

Tree balancing. Even with continuous constraint solving, co-contextual FJ as defined in Section 9.2 still does not yield satisfactory incremental performance. The reason is that the syntax tree is deformed due to the root node, which consists of a sequence of *all* class declarations in the program. Thus, the root node has a branching factor only bound by the number of classes in the program, whereas the rest of the tree has a relative small branching factor bound by the number of arguments to a method. Since incremental type checking recomputes each step from the changed node to the root node, the type checker would have to repeat discharging class requirements at the root node after every code change, which would seriously impair incremental performance.

To counter this effect, we apply tree balancing as our final optimization. Specifically, instead of storing the class declarations as a sequence in the root node, we allow sequences

of class declarations to occur as inner nodes of the syntax tree:

$$L ::= \bar{L} \mid \text{class } C \text{ extends } D \{ \bar{C} \ \bar{f}; \ K \ \bar{M} \}$$

This allows us to layout a program's class declarations structurally as in $((((C_1 \ C_2) \ C_3) \ (C_4 \ C_5)) \ (C_6 \ C_7))$, thus reducing the costs for rechecking any path from a changed node to the root node. As part of this optimization, to satisfy requirements of classes that occur in different tree nodes such as C_1 and C_6 , we also needed to propagate *class facts* such as actual method signatures upwards. As consequence, we can now link classes in any order without changing the type checking result.

We have implemented an incremental co-contextual FJ type checker in Scala using the optimizations described here. In the following section, we present our runtime performance evaluation.

9.5. Performance Evaluation

We have benchmarked the initial and incremental runtime performance of co-contextual FJ implementation. This evaluation is intended to confirm the feasibility and potential of co-contextual FJ for incremental type checking.

9.5.1. Evaluation on synthesized FJ programs

Input data. We synthesized FJ programs with 40 root classes that inherit from Object. Each root class starts a binary tree in the inheritance hierarchy of height 5. Thus, each root-class hierarchy contains 31 FJ class declarations because the number of nodes in a balanced tree of height³ h is $2^h - 1$. In total, our synthesized programs have $31 * 40 + 3 = 1243$ class declarations, since we always require classes for natural numbers Nat, Zero, and Succ.

Each class has at least a field of type Nat and each class has a single method that takes no arguments and returns a Nat. We generated the method body according to one of three schemes:

- *AccumSuper*: The method adds the field's value of this class to the result of calling the method of the super class.
- *AccumPrev*: Each class in root hierarchy $k > 1$ has an additional field that has the type of the class at the same position in the previous root hierarchy $k - 1$. The method adds the field's value of this class to the result of calling the method of the class at the same position in the previous root hierarchy $k - 1$, using the additional field as receiver object.
- *AccumPrevSuper*: Combines the other two schemes; the method adds all three numbers.

³Where height is 1 for a single root node

9. Co-contextual Featherweight Java

We also varied the names used for the generated fields and methods:

- *Unique*: Every name is unique.
- *Mirrored*: Root hierarchies use the same names in the same classes, but names within a single root hierarchy are unique.
- *Override*: Root hierarchies use different names, but all classes within a single root hierarchy use the same names for the same members.
- *Mir+Over*: Combines the previous two schemes, that is, all classes in all root hierarchies use the same names for the same members.

For evaluating the incremental performance, we invalidate the memoized results for the three `Nat` classes. This is a critical case because all other classes depend on the `Nat` classes and a change is traditionally hard to incrementalize.

Experimental setup. First, we measured the wall-clock time for the initial check of each program using our co-contextual FJ implementation. Second, we measured the wall-clock time for the incremental reanalysis after invalidating the memoized results of the three `Nat` classes. Third, we measured the wall-clock time of checking the synthesized programs on `javac` and on a straightforward implementation of contextual FJ for comparison. Contextual FJ is the standard FJ described in Section 8.1, that uses contexts and class tables during type checking. Our implementation of contextual FJ is up to 2-times slower than `javac`, because it is not production quality. We used `ScalaMeter`⁴ to take care of JIT warm-up, garbage-collection noise, etc. All measurements were conducted on a 3.1GHz duo-core MacBook Pro with 16GB memory running the Java HotSpot 64-Bit Server VM build 25.111-b14 with 4GB maximum heap space. We confirmed that confidence intervals were small.

Results. We show the measurement results in table 9.1. All numbers are in milliseconds. We also show the speedups of initial and incremental run of co-contextual type checking relative to both `javac` and contextual type checking.

As this data shows, the initial performance of co-contextual FJ is subpar: The initial type check takes up to 68-times and 61-times longer than using `javac` and a standard contextual checker respectively.

However, co-contextual FJ consistently yields high speedups for incremental checks. Incremental co-contextual FJ is up to 6-times and 8-times faster than `javac` and a standard contextual checker respectively. In fact, it only takes between 3 and 21 code changes until co-contextual type checking is faster overall. In an interactive code editing session where every keystroke or word could be considered a code change, incremental co-contextual type checking will quickly break even and outperform a contextual type checker or `javac`.

The reason that the initial run of co-contextual FJ induces such high slowdowns is because the occurrence of class requirements is far removed from the occurrence of the corresponding class facts. This is true for the `Nat` classes that we merge with the

⁴<https://scalameter.github.io/>

Super	javac / contextual	co-contextual init	co-contextual inc
unique	70.00 / 93.99	3117.73 (0.02x / 0.03x)	23.44 (2.9x / 4x)
mirrored	68.03 / 88.73	1860.18 (0.04x / 0.05x)	15.17 (4.5x / 6x)
override	73.18 / 107.83	513.44 (0.14x / 0.21x)	16.92 (4.3x / 6x)
mir+over	72.64 / 132.09	481.07 (0.15x / 0.27x)	16.60 (4.4x / 8x)
Prev	javac / contextual	co-contextual init	co-contextual inc
unique	82.16 / 87.66	3402.28 (0.02x / 0.02x)	23.43 (3.5x / 4x)
mirrored	81.19 / 84.94	2136.42 (0.04x / 0.04x)	15.46 (5.3x / 5x)
override	81.51 / 120.60	840.14 (0.09x / 0.14x)	17.37 (4.7x / 7x)
mir+over	79.71 / 120.46	816.16 (0.09x / 0.15x)	16.61 (4.8x / 7x)
PrevSuper	javac / contextual	co-contextual init	co-contextual inc
unique	93.12 / 104.03	6318.69 (0.01x / 0.02x)	26.26 (3.5x / 4x)
mirrored	95.41 / 100.00	5014.12 (0.02x / 0.02x)	15.71 (6.1x / 6x)
override	92.88 / 130.01	3601.44 (0.03x / 0.04x)	17.35 (5.4x / 7x)
mir+over	93.37 / 126.57	3579.90 (0.03x / 0.04x)	16.61 (5.6x / 8x)

Table 9.1.: Performance measurement results with $k = 40$ root classes in **Milliseconds**. Numbers in parentheses indicate speedup relative to (javac/contextual) base lines.

synthesized code at the top-most node as well as for dependencies from one root hierarchy to another one. Therefore, the type checker has to propagate and merge class requirements for a long time until finally discovering class facts that discharge them. We conducted an informal exploratory experiment that revealed that the performance of the initial run can be greatly reduced by bringing requirements and corresponding class facts closer together. On the other hand, incremental performance is best when the changed code occurs as close to the root node as possible, such that a change entails fewer rechecks.

9.5.2. Evaluation on real Java program

Input data. We conduct an evaluation for our co-contextual type checking on realistic FJ programs. We wrote about 500 SLOCs in Java, implementing purely functional data structures for binary search trees and red black trees. In the Java code, we only used features supported by FJ and mechanically translated the Java code to FJ. For evaluating the incremental performance, we invalidate the memoized results for the three Nat classes as in the experiment above.

Experimental setup. Same as above.

Results. We show the measurements in milliseconds for the 500 lines of Java code.

javac / contextual	co-contextual init	co-contextual inc
14.88 / 3.74	48.07 (0.31x / 0.08x)	9.41 (1.6x / 0.39x)

9. *Co-contextual Featherweight Java*

Our own non-incremental contextual type checker is surprisingly fast compared to `javac`, and not even our incremental co-contextual checker gets close to that performance. When comparing `javac` and the co-contextual type checker, we observe that the initial performance of the co-contextual type checker improved compared to the previous experiment, whereas the incremental performance degraded. While the exact cause of this effect is unclear, one explanation might be that the small input size in this experiment reduces the relative performance loss of the initial co-contextual check, but also reduces the relative performance gain of the incremental co-contextual check.

10. Co-Contextual Featherweight Java with Generics

In this chapter, we extend FJ with generics (FGJ) and co-contextualize it. Generic types allow programmers to implement algorithms in a type-safe manner and avoid the usage of class casts. For example, to provide generic list type structures that can be used for all types of list elements. Therefore, FGJ is a very powerful language extension to FJ, which makes the programmers life easier. However, it is challenging to co-contextualize FGJ because of the generic types and the impact they have in performing the operations of merging and removing class table requirements.

We start by describing the traditional FGJ, i.e., the types, syntax, contexts and judgements used while type checking, and then provide the contextual typing rules. Next, we present the co-contextual structures for FGJ and describe the syntax and the changes to the requirements sets and judgements. Then, extend the operations on the class table requirements in the presence of generic types. Finally, we construct the co-contextual typing rules for FGJ by applying dualism.

10.1. Featherweight Java with Generics

FGJ was introduced in the same paper as FJ [IK01]. In FGJ, classes and methods have generic type parameters, represented by angle braces (< and >). Hence, a single declaration of a method or class corresponds to a set of related methods with different signatures or a set of related types, respectively, depending on the usages of the generic types. More specifically, generic methods are those methods that are written as a single method declaration and can be called with arguments of different types. The type checker ensures the correctness of whichever type is used. To give an intuition of the generic classes and methods let us consider the following example:

```
class Pair <X extends Object, Y extends Object> extends Object {
    X fst;
    Y snd;
    Pair(X fst, Y snd) {
        super(); this.fst=fst; this.snd=snd;
    }
    <Z extends Object> Pair<Z,Y> setfst(Z newfst) {
        return new Pair<Z,Y>(newfst, this.snd);
    }
}
```

The class *Pair* is parametrized over the generic types X and Y . The fields of this class *fst* and *snd* have as types X and Y , respectively. Different objects of *Pair* can have different instantiations of X and Y . A generic type can be substituted to different types. Consequently, *fst* and *snd* have different types corresponding to X and Y . For example, *fst* can be used in the program with types *Int* or *String*, obtained from $(\text{new Pair}\langle\text{Int}, \text{Int}\rangle(1, 2)).\text{fst}$ or $(\text{new Pair}\langle\text{String}, \text{String}\rangle(\text{hello}, \text{world})).\text{fst}$, respectively.

Similar to the class *Pair*, the method *setfst* is parameterized over the generic type Z . Depending on the usage of Z in method invocations the method will return different result types. For example the invocation $(\text{new Pair}\langle\text{Int}, \text{Int}\rangle(1, 2)).\text{setfst}\langle\text{Int}\rangle(2)$, yields a result of type $\text{Pair}\langle\text{Int}, \text{Int}\rangle$, while $(\text{new Pair}\langle\text{Int}, \text{Int}\rangle(1, 2)).\text{setfst}\langle\text{String}\rangle(\text{hello})$ yields a result of type $\text{Pair}\langle\text{String}, \text{Int}\rangle$.

$T ::= X \mid N$	Types
$N ::= C \langle \overline{T} \rangle$	Nonvariable types
$L ::= \text{class } C \langle \overline{X} \triangleleft \overline{N} \rangle \triangleleft N \{ \overline{T} \ \overline{f}; \ K \ \overline{M} \}$	class declaration
$K ::= C(\overline{T} \ \overline{f}) \{ \text{super}(\overline{f}); \ \text{this}.\overline{f} = \overline{f} \}$	constructor
$M ::= \langle \overline{X} \triangleleft \overline{N} \rangle \ T \ m(\overline{T} \ \overline{x}) \{ \text{return } e; \}$	method declaration
$e ::= x \mid \text{this} \mid e.f \mid e.m \langle \overline{T} \rangle(\overline{e}) \mid \text{new } N(\overline{e}) \mid (N)e$	expression
$\Gamma ::= \emptyset \mid \Gamma; x : C \mid \Gamma; \text{this} : C$	typing contexts
$\Delta ::= \emptyset \mid \Delta; X \triangleleft N$	bounds

Figure 10.1.: FGJ syntax, typing contexts, and bounds.

In the following, we recap the FJ syntax and typing rules required to support generics. The syntax for FGJ is illustrated in Figure 10.1. The types of FGJ extend the FJ types with generic types, where X denotes type variables and N ranges over nonvariable types. N represents classes, which are parametrized over other types; type variables or nonvariable types. L represents class declarations. Classes are parametrized over generic types, i.e., bounded type variables. These type variables can be used as types for the member fields of the given class, as shown in the example above (X *fst*). The relation between type variables \overline{X} and their bounds \overline{N} is represented via the symbol \triangleleft , which stands for **extends** in FJ. Similar to classes, methods (M) are parametrized over type variables. Generic type parameters ($\langle \overline{X} \triangleleft \overline{N} \rangle$) of a method can be used as part of the method signature, serving as types of the method parameters. Regarding expressions, there is a change at method invocation and object creation. In case of method invocation, while calling a generic method its generic parameters are instantiated from type variables to concrete types. In the case of the **new** expression, the class N is a nonvariable type, i.e., if the class has generic parameters, then while calling the constructor of that class the generic parameters are instantiated.

In FGJ, the type variables used to parametrize classes and methods are bounded. This information is stored to an additional context to Γ and class table CT , which is denoted by Δ . Bounds of the types are retrieved using the following auxiliary function:

$$\text{bound}_\Delta(X) = \Delta(X)$$

$\text{bound}_\Delta(N) = N$

The bound function applied to a type variables returns the bounds of the type variables by looking them up in Δ . However, bound applied to a nonvariable type returns the type itself.

FGJ introduces a new judgement for type well-formedness $\Delta \vdash T \text{ ok}$, in addition to the typing and subtyping judgements, which are used in FJ.

$$\begin{array}{c}
 \text{WF-OBJECT} \frac{}{\Delta; CT \vdash \text{Object ok}} \qquad \text{WF-VAR} \frac{X \in \text{dom}(\Delta)}{\Delta; CT \vdash X \text{ ok}} \\
 \\
 \text{WF-CLASS} \frac{\Delta; CT \vdash \bar{T} \text{ ok} \quad \Delta \vdash \bar{T} <: [\bar{T}/\bar{X}]\bar{N} \quad \text{extends}(C <\bar{X} \triangleleft \bar{N}>, CT) = N}{\Delta; CT \vdash C <\bar{T}> \text{ ok}}
 \end{array}$$

Figure 10.2.: Well-Formedness Rules.

Figure 10.2 illustrates the rules for well-formed types. In contrast to the original well-formedness rules, we make the class table CT explicit. If the declaration of a class C begins *class* $C <\bar{X} \triangleleft \bar{N}>$, then the type $C <\bar{T}>$ is well-formed if \bar{X} substituted by \bar{T} respects the bounds \bar{N} i.e., if $\bar{T} <: [\bar{T}/\bar{X}]\bar{N}$. We write $\Delta \vdash T \text{ ok}$ if type T is well formed in the context Δ .

Subtyping is as in FJ, with the difference that the type checker uses the information provided by Δ in addition to the information provided by the class hierarchies in a class table to solve subtyping constraints.

In the following, we describe the typing rules of FGJ, considering only the changes we did to the contextual typing rules of FJ to support FGJ. All typing judgements of FGJ, in contrast to FJ, have an additional context to Γ , CT , which is Δ to keep track of the bounded types.

10.1.1. Featherweight Java with Generics Typing Rules

We do not describe the typing rules for variables, casting and programs because they do not change from the corresponding ones presented in FJ, except that they use the additional context Δ .

The rule GT-FIELD is used for checking the field access. This rule is similar to the rule T-FIELD in FJ, with one change. The auxiliary function `field` is used to retrieve the type of the accessed field, which looks up the field type in the bound of the expressions type and not in the type itself.

The type checker uses the rule GT-INVK to type check method invocation in the presence of generic types. The type checker starts with type checking the expression e , giving as

GT-FIELD	$\frac{\Delta; \Gamma; CT \vdash e : T_e \quad \text{field}(f_i, \text{bound}_\Delta(T_e), CT) = T_i}{\Delta; \Gamma; CT \vdash e.f_i : T_i}$
GT-INVK	$\frac{\Delta; \Gamma; CT \vdash e : T_e \quad \text{mtype}(m, \text{bound}_\Delta(T_e), CT) = \langle \bar{Y} \triangleleft \bar{P} \rangle \bar{U} \rightarrow U \quad \Delta \vdash \bar{V} \text{ ok} \quad \Delta \vdash \bar{V} <: [\bar{V}/\bar{Y}] \bar{P} \quad \Delta; \Gamma; CT \vdash \bar{e} : \bar{S} \quad \Delta \vdash \bar{S} <: [\bar{V}/\bar{Y}] \bar{U}}{\Delta; \Gamma; CT \vdash e.m \langle \bar{V} \rangle (\bar{e}) : [\bar{V}/\bar{Y}] U}$
GT-NEW	$\frac{\Delta \vdash N \text{ ok} \quad \text{fields}(N, CT) = N.\text{init}(\bar{T}) \quad \Delta; \Gamma; CT \vdash \bar{e} : \bar{S} \quad \Delta \vdash \bar{S} <: \bar{T}}{\Delta; \Gamma; CT \vdash \text{new } N(\bar{e}) : N}$
GT-METHOD	$\frac{\Delta = \bar{X} <: \bar{N}, \bar{Y} <: \bar{P} \quad \Delta; \bar{x} : \bar{T}; \text{this} : C \langle \bar{X} \rangle; CT \vdash e : S \quad \text{extends}(C \langle \bar{X} \rangle \bar{N}, CT) = N \quad \Delta \vdash \bar{T}, T, \bar{P} \text{ ok} \quad \Delta \vdash S <: T \quad \text{override}(m, N, \langle \bar{Y} \rangle \bar{T} \rightarrow T, CT)}{C \langle \bar{X} \rangle \bar{N}; CT \vdash \langle \bar{Y} \rangle \bar{T} \ m(\bar{T} \ \bar{x}) \{ \text{return } e \} \text{ OK}}$
GT-CLASS	$\frac{\bar{X} <: \bar{N} \vdash \bar{N}, N, \bar{T} \text{ ok} \quad \text{fields}(N, CT) = N.\text{init}(\bar{U}) \quad K = C(\bar{U} \ \bar{g}, \bar{T}' \ \bar{f}) \{ \text{super}(\bar{g}); \text{this}.\bar{f} = \bar{f} \} \quad C \langle \bar{X} \rangle \bar{N}; CT \vdash \bar{M} \text{ OK}}{CT \vdash \text{class } C \langle \bar{X} \rangle \bar{N} \triangleleft N \ \{ \bar{T} \ \bar{f}; K \ \bar{M} \} \text{ OK}}$
GT-PROGRAM	$\frac{CT = \bigcup_{L' \in \bar{L}} (\text{addExt}(L') \cup \text{addCtor}(L') \cup \text{addFs}(L') \cup \text{addMs}(L')) \quad (CT \vdash L' \text{ OK})_{L' \in \bar{L}}}{\bar{L} \text{ OK}}$

Figure 10.3.: Typing rules of Featherweight Java with Generics.

result the type T_e . Then, it looks up the signature of the method m in the class table CT , as member of the bound of T_e . The result of the look up is a generic type. That is, part of the method signature are also its generic parameters. These parameters are instantiated while invoking the method m . The instantiated types should be subtypes of the bounds of the generic type parameters ($\bar{V} <: [\bar{V}/\bar{Y}] \bar{P}$). Finally, the type checker ensures that instantiated types \bar{V} are *ok* given the context Δ .

The rule GT-NEW is used to type check object creation. The type checker starts with type checking the class N of the object given the context Δ . The type checker uses the auxiliary function *fields* as in FJ to obtain the constructor signature, which gives as result the constructor clause *init* for the class N . Finally, it ensures that the signature of the object is a subtype of the actual constructor signature.

Next, we consider the rule GT-METHOD used to type check method declarations. In the case of generics, methods are parametrized over bounded type variables $\langle \bar{Y} \rangle \bar{P}$. In

addition, the class C , where the method is type checked in, is parametrized $\langle \overline{X} \triangleleft \overline{N} \rangle$. The context Δ is created using these two sets of bounded type variables; $\Delta = \overline{X} <: \overline{N}, \overline{Y} <: \overline{P}$. Given this context Δ , the parameter and return types of the method m , and the generic parameters of the method are type checked to be *ok*. Next, the type checker encounters the body e_0 of the method and ensures that its type is a subtype of the return type of m . Finally, it checks that the overriding of the method m is valid. The overriding is checked if a declaration of m exists also in superclasses of C . In FJ, a method m must have the same signature in all its declarations, in the class C or its superclasses. However, FGJ allows that the return type of the overriding method m in C is a subtype of the return type of the overridden method m in superclasses of C (covariant return types). Parameter types must be the same in all declarations of m . To ensure a proper overriding, the type checker uses the override function.

Finally, we consider the rule GT-CLASS for class typing. This rule has the same judgements as the corresponding rule in FJ. The only difference is that classes have generic parameters, which construct the Δ . Then, the type checker ensures that all generic parameters, the superclass N of the class C , and the field types are *ok* under the context Δ .

10.2. Co-Contextual Structures for Featherweight Java with Generics

In this section, we describe the required steps to construct a co-contextual type checker for FGJ. We use the dualism technique to remove the class table and all other contexts, and instead introduce the corresponding dual requirements. FGJ in particular introduces the bounds context Δ in addition to the class table CT and the typing context Γ . We introduce a new set requirements dual to Δ and extend the operations on requirements, with duals to the operations on Δ . Moreover, FGJ uses the well-formedness judgement, in addition to the judgements used in FJ. We construct the co-contextual well-formedness rules applying our dualism technique.

In the following, we first extend the previous co-contextual formulation of FJ in terms of constraints. We then discuss the new set of requirements and judgement.

10.2.1. Co-contextual Constraints

The set of constraints used in co-contextual FJ is extended with new constraints to support generics, which are shown below:

$$s ::= \dots \mid T = \{X \mapsto T\}T \mid T = \{U \mapsto T\}T \mid T <: \{X \mapsto T\}T \mid T <: \{U \mapsto T\}T \mid (T < T)_{opt}$$

We require new constraints of the form $T_1 = \{X \mapsto T_2\}T_3$ and $T_1 <: \{X \mapsto T_2\}T_3$. The latter expresses that T_1 is subtype of substituting T_2 for X in T_3 . We define similar constraints for substituting unification variables. The subtyping substitution constraint $T_1 <: \{X \mapsto T_2\}T_3$ is equivalent to the subtyping constraint $\overline{S} <: [\overline{V}/\overline{Y}]\overline{U}$ in FGJ. These

constraints are similar to the constraints introduced in parametric polymorphism (Chapter 4) and, hence, are not discussed in more detail in this chapter.

The last constraint $(T <: T)_{opt}$ is introduced to ensure a valid method overriding in the presence of generics, which allows methods to have covariant return types. We will discuss this constraint in more detail, when showing the co-contextual typing rule for method declarations.

However, the types of co-contextual FGJ include generic types in addition to the types described for co-contextual FJ. Generic types are the same as for FGJ, described in the previous section.

In the rest of this section, we construct and describe the additional set of requirements and judgement used by co-contextual FGJ. We apply our technique (dualism) to build a co-contextual FGJ type checker. Therefore, all contexts are removed; instead we introduce the dual structure of requirements. As shown above, FGJ uses the contexts Γ and Δ , and the class table CT while type checking. To construct a co-contextual type system, we remove the context Γ and class table CT , instead we introduce the dual structures of context and class table requirements correspondingly, as previously discussed for co-contextual FJ. The context requirements have the structure and operations as for co-contextual FJ. The class table requirements have the same structure as the class table requirements used in co-contextual FJ, with the difference that classes and methods are parametrized. Moreover, the operations of merging and removing requirements are adapted for the generic types, which we will describe in the next section.

10.2.2. Bounded requirements

In addition to Γ and CT , FGJ uses the context Δ for type bounds. Therefore, we introduce the dual structure of bounded requirements BR , which are a mapping from types to types. These requirements as for the other sets of requirements have the operations of merging and removing. However, the merge and remove cannot be performed instantly, as in case of context requirements because bounded requirements are requirements on types. Because of nominal typing, as for the class table requirements, we do not know the actual type variables and their bounds. Therefore, we introduce conditions for these requirements. These conditions operate similar to the conditions in class table requirements. The structure of bounded requirements is shown below:

$$BR = \{(T \rightsquigarrow T, cond)\}$$

The operations on the bounded requirements are merge and remove. Merge of two sets of bounded requirements results in a new set of bounded requirements and a set of constraints:

$$\text{merge}_{BR}(BR_1, BR_2) = BR_m|_{S_m}$$

where the sets CR_m and S_m are shown in Figure 10.4. Two bounded requirements are merged, if the required types variables are the same. The co-contextual type checker does not know the actual type variables because of nominal typing. The types T_1 and T_2 of the

$$\begin{aligned}
 BR_m &= \{(T_1 \rightsquigarrow T'_1, \text{cond}_1 \cup (T_1 \neq T_2)) \cup (T_2 \rightsquigarrow T'_2, \text{cond}_2 \cup (T_1 \neq T_2)) \\
 &\quad \cup (T_1 \rightsquigarrow T'_1, \text{cond}_1 \cup \text{cond}_2 \cup (T_1 = T_2)) \\
 &\quad \mid (T_1 \rightsquigarrow T'_1, \text{cond}_1) \in BR_1 \wedge (T_2 \rightsquigarrow T'_2, \text{cond}_2) \in BR_2\} \\
 S_m &= \{(T'_1 = T'_2 \text{ if } T_1 = T_2) \mid (T_1 \rightsquigarrow T'_1, \text{cond}_1) \in BR_1 \wedge (T_2 \rightsquigarrow T'_2, \text{cond}_2) \in BR_2\}
 \end{aligned}$$

Figure 10.4.: Merge operation on bounded requirements.

bounded requirements could still be unification variables U . Consequently, we cannot decide whether the two requirements can be merged or not. To solve this, we add type equalities and inequalities to the conditions to cover the cases when the type variables of two requirements are the same or not, respectively $T_1 = T_2$ and $T_1 \neq T_2$. The first case indicates that the two requirements can be merged.

Adding a declaration for a bounded type variable is dual to removing the bounded requirements corresponding to that type variable.

$$\begin{aligned}
 \text{removeB}(X \triangleleft N, BR) &= BR'|_S \\
 \text{where } BR' &= \{(T \rightsquigarrow T', \text{cond} \cup (T \neq X)) \mid (T \rightsquigarrow T', \text{cond}) \in BR\} \\
 S &= (T' = N \text{ if } T = X) \mid (T \rightsquigarrow T', \text{cond}) \in BR \\
 \text{removeBR}(\overline{X} \triangleleft \overline{N}, BR) &= BR'|_S \\
 \text{where } BR' &= \{BR_r \mid X \triangleleft N \in \overline{X} \triangleleft \overline{N} \wedge \text{removeB}(X \triangleleft N, BR) = BR_r|_{S_r}\} \\
 S &= \{S_r \mid X \triangleleft N \in \overline{X} \triangleleft \overline{N} \wedge \text{removeB}(X \triangleleft N, BR) = BR_r|_{S_r}\}
 \end{aligned}$$

Figure 10.5.: Remove operation on bounded requirements.

Figure 10.5 shows the remove operation for a given bounded variable $X \triangleleft N$. All bounded requirements in BR are considered and their conditions are updated. The type checker does not know the actual type variable corresponding to T , therefore, it updates the condition with the type inequality $T \neq X$. Moreover, we generate the conditional constraint S . Thus, if the actual type of T is learned to be X , then we have satisfied the requirement. Consequently, the bounded requirement is discharged because its condition $\text{cond} \cup (T \neq X)$ is unsatisfiable; $(T \neq X)$ does not hold.

10.2.3. Co-Contextual Well-Formedness Rules

FGJ uses an additional well-formedness judgement. To systematically co-contextualize FGJ, we have to co-contextualize also the rules for well-formed types. We construct co-contextual well-formedness rules, which are dual to the rules shown in Figure 10.2.

The contextual well-formedness rules take as input the context Δ , class table CT and do not have an output. They only ensure that the types are *ok*. Applying dualism, we remove the context Δ and the class table CT . The co-contextual rules output sets of bounded requirements, class table requirements, and constraints. Constraints are generated because our co-contextual formulations are constraints-based.

$$\begin{array}{c}
 \text{CWF-OBJECT} \frac{}{\vdash \text{Object ok} \mid \emptyset \mid \emptyset \mid \emptyset} \qquad \text{CWF-VAR} \frac{U \text{ is fresh}}{\vdash X \text{ ok} \mid \emptyset \mid \emptyset \mid X \rightsquigarrow U} \\
 \\
 \text{CWF-CLASS} \frac{\begin{array}{c} \vdash T \text{ ok} \mid S_t \mid CR_t \mid BR_t \\ S = \{T <: \{U_x \mapsto T\} U_n\} \quad CR_n = (C <\overline{U}_x \triangleleft \overline{U}_n>.\text{extends} : U_d) \\ CR_{S_m} = \text{merge}_{CR}(CR_t, CR_n) \end{array}}{\vdash C <\overline{T}> \text{ ok} \mid S_t \cup \overline{S} \cup S_m \mid CR \mid BR_t}
 \end{array}$$

Figure 10.6.: Co-Contextual Well-Formedness Rules.

The rule CWF-OBJECT is straightforward. The rule CWF-VAR checks the well-formedness of the type variable X . For X to be well-formed it should exist a bound of X in Δ . However, the co-contextual type checker cannot check the existence of X in Δ . To ensure that a declaration of X will be found in Δ , we generate a fresh unification variable and assign it to X . $X \rightsquigarrow U$ is added to the bounded requirements. The rule CWF-CLASS checks the well-formedness of classes. Dual to looking up a class declaration in CT is introducing a requirements for the **extends** clause. The unification variables \overline{U}_x and \overline{U}_n are placeholders for the type variables \overline{X} and their bounds \overline{N} , respectively.

10.3. Operations on Class tables Requirements

In the following, we describe the merge and remove operations for class table requirements with generic types.

10.3.1. Merge Operation

In co-contextual FJ, the merge of two requirements, which have the same field name or method symbol, is performed if the receiver classes of these two requirements have the same type. However, in FGJ, classes are parametrized and they are considered to be the same if and only if they have the same name and equal types on the generic parameters. This affects the merging of the class table requirements.

To illustrate the merge operation consider the following two field requirements $CR_1 = (T_1.f : T_f, cond_1)$ and $CR_2 = (T_2.f : T'_f, cond_2)$. Both requirements operate on the equally-named field f .

$$\begin{aligned}
 CR_m = & \{(T_1.f : T_f, cond_1 \cup (T_1 \neq T_2)) \cup (T_2.f : T'_f, cond_2 \cup (T_1 \neq T_2)) \\
 & \cup (T_1.f : T_f, cond_1 \cup cond_2 \cup (T_1 = T_2)) \\
 & \mid (T_1.f : T_f, cond_1) \in CR_1 \wedge (T_2.f : T'_f, cond_2) \in CR_2\} \\
 S_m = & \{(T_f = T'_f \text{ if } T_1 = T_2) \\
 & \mid (T_1.f : T_f, cond_1) \in CR_1 \wedge (T_2.f : T'_f, cond_2) \in CR_2\}
 \end{aligned}$$

 Figure 10.7.: Merge operation of field requirements CR_1 and CR_2 .

Figure 10.7 illustrates the result of merging the two field requirements. For the merge to succeed, both requirements must have the same receiver classes. However, the actual type of the receiver classes is not known because of the nominal typing. Hence, we generate type equalities and inequalities to foresee the cases when the receiver classes are the same or not ($T_1 = T_2$ and $T_1 \neq T_2$), which are added to the conditions of the requirements. If the receiver classes are the same then the requirements are merged and a conditional constraint is generated, stating that the types of these two fields should be equal. The merge operation is similar to the merging in co-contextual FJ. However, in FGJ, we have to take care when we compare the receiver classes. In addition to the names, we have to compare the generic parameters of the classes. If the receiver classes are objects, then the instances of the generic parameters of the two receivers must be the same. Otherwise, the type variables of the generic parameters of the two receiver classes must be the same.

10.3.2. Remove Operation

We now adapt the remove operation for generic types. In the case of generics, a declared field can have as type a ground type as in FJ or a type variable. This type variable is one of the generic types of a parametrized class that declares the given field as a member. Hence, we have to consider these two cases while applying the remove operation for field clauses. Likewise, in the case of methods, the parameter and return types can be either ground types or type variables. These type variables are part of the generic types of the given method or the class, declaring the method as member. To illustrate the remove operation with generic types, we consider the remove of field requirements.

Figure 10.8 shows add operation for field clauses to the class table CT and the corresponding dual operation of removing field requirements. The field clause has the same structure as in FJ, namely $C <\overline{X} \triangleleft \overline{N}>.f : T$. The remove operation finds the field requirements that have the same name as the declared field under scrutiny. Because of nominal typing, the requirement $(T'.f : T_f, cond)$ cannot be instantly removed from the requirements set given the declaration of the field f . That is, the type of the receiver class T' is not known and we cannot decide whether it is equal to C , or not. Therefore, we update the condition by introducing the inequality $(T'.name \neq C)$, which will indicate

$$\begin{aligned}
\text{adds}(C < \overline{X} \triangleleft \overline{N} >, \overline{T} \ \overline{f}, CT) &= \overline{C < \overline{X} \triangleleft \overline{N} >.f : T \cup CT} \\
\text{removeF}(C < \overline{X} \triangleleft \overline{N} >, T \ f, CR) &= CR'|_S \\
\text{where } CR' &= \{(T'.f : T_f, \text{cond} \cup (T'.\text{name} \neq C)) \mid (T'.f : T_f, \text{cond}) \in CR\} \\
&\quad \cup (CR \setminus \overline{(T'.f : T_f, \text{cond})}) \\
S &= \begin{cases} T = T_f & \text{if } T'.\text{name} = C \\ T = \text{getTypeF}(C < \overline{X} \triangleleft \overline{N} >, T', T_f) & \text{if } T'.\text{name} = C \text{ otherwise} \end{cases} \\
\text{removeFs}(C, \overline{T} \ \overline{f}, CR) &= CR'|_S \\
\text{where } CR' &= \{CR_m \mid (T \ f) \in \overline{T} \ \overline{f} \wedge \text{removeF}(C < \overline{X} \triangleleft \overline{N} >, T \ f, CR) = CR_m|_{S_m}\} \\
S &= \{S_m \mid (T \ f) \in \overline{T} \ \overline{f} \wedge \text{removeF}(C < \overline{X} \triangleleft \overline{N} >, T \ f, CR) = CR_m|_{S_m}\}
\end{aligned}$$

Figure 10.8.: Add and remove operations of field clauses.

that the requirement can be discharged if its condition is unsatisfiable. At the point in time, when we will know the actual type of T' , we apply substitution to the condition and can compare the two types.

However, in this comparison we ignore the generic parameters and only check if classes have the same name. We do so because the receiver class of the requirement contains parameters for the generic types. These parameters should be subtypes of the bounds of the generic types, but this check is done in Δ . While comparing the receiver class of f requirement and the class where f is declared to, we consider only their names. This name-based comparison is achieved via the auxiliary function `name`, e.g., $C < \overline{X} \triangleleft \overline{N} >.\text{name} = C$. As a result, if $T'.\text{name} = C$, then we have found a corresponding declaration for the required field and know its actual type. Consequently, the requirement is removed. For a requirement to be removed from the requirements set its condition should be unsatisfiable, i.e., at least one of the type equalities or inequalities in the condition should not hold.

In order to deduce the actual type of the required field the conditional constraint S is generated. This constraint ensures that the receiver class of the required field has the same name as the class, where the declaration of this field belongs to. Depending on the type of the declared field f , we distinguish two cases, the field f has 1) a ground type, or 2) a type variable. For any of these cases to hold, the condition $T'.\text{name} = C$ should hold. Then, the cases are scrutinized. In the first case, a conditional equality constraint is generated, as for co-contextual FJ and no further changes are required. In the second case, we use an auxiliary function to deduce the actual type of the required field, as shown below:

$$\begin{aligned}
&\text{getTypeF}(c : C < \overline{X} \triangleleft \overline{N} >, \text{obj} : T, \text{tf} : X) : \text{Type} = \{ \\
&\quad \text{if } (T == C < \overline{X} \triangleleft \overline{N} >) \\
&\quad \quad X
\end{aligned}$$


```

else {
  T = C < $\overline{T}$ >
  for(i ← 0 until  $\overline{X}$ .length) {
    if (  $\overline{X}(i)$  == tF)
      return  $\overline{T}(i)$  }
  }
}

```

A field f is accessed by a class, or a class instance. Consequently, we distinguish two cases; 1) a field is accessed by a class which has generic parameters, e.g., in case of method declarations, or 2) a field is accessed by an object of a class. In the first case, we only return the type variable of the declared field f , which will be the type of the required field f . In the second case, we ensure that the required field has a type, which is an instantiation of its corresponding declared type variable. An object has instances of the generic parameters. To deduce the specific type of a field, we have to know the position of the generic class parameter that corresponds to the type variable of the declared field f . This is realized via the function `getTypeF`, which gets the position of the type variable corresponding to f in \overline{X} , then it applies this position to the sequence of instances \overline{T} of an object of a class C . The constraint S is solved when we know the actual type of T' .

Let us illustrate the remove operation given the field access `new Pair<Int, Int>(1, 2).fst` from the example in Section 10.1. In our co-contextual setting, the dual of accessing the field `fst` is introducing a new requirement for the field `fst`. The resulting type of the constructor call is `Pair<Int, Int>`, which is the receiver class of the required field, i.e., `Pair<Int, Int>.fst : U`. U is a fresh unification variable because we do not know the actual type of `fst`, until we encounter a declaration corresponding to it. Given the declaration `(Pair<X< Object, Y< Object>. fst : X)`, we can apply `remove` on the field requirement for `fst`, which generates the constraint S . The actual type of `fst` is not ground. Therefore, S is $U = \text{getTypeF}(\text{Pair} <X < \text{Object}, Y < \text{Object}>, \text{Pair} <\text{Int}, \text{Int}>, X)$. The result from applying the function `getTypeF` is $U = \text{Int}$ and the condition `Pair<Int, Int>.name = Pair` holds. As a result, the type of the accessed field `fst` is `Int`.

10.4. Co-Contextual Featherweight Java with Generics Typing Rules

In this section, we derive the typing rules for co-contextual FGJ. The co-contextual typing rules are constraint-based and do not have contexts and a class table. Instead, they generate requirements. The translation is systematic from contextual to co-contextual FGJ and we use dualism to enable this translation. The complete set of type rules for co-contextual FGJ is shown in Figure 10.9.

The rule `GTC-FIELD` is used to type check field accesses. The type checker starts with the receiver expression e . The contextual rule for field access `GT-FIELD` uses the function `bound` to get the bound of T_e . The dual to looking up the bound of a type is introducing a bounded requirement. However, this function applied to a nonvariable type returns the type itself and not a look up in Δ . In the co-contextual setting, we do not know whether

GTC-FIELD	$\frac{e : T_e \mid S_e \mid R_e \mid CR_e \mid BR_e \quad BR _{S'} = \text{merge}_{BR}(BR_e, (T_e \rightsquigarrow U)_{opt}) \quad CR _{S_f} = \text{merge}_{CR}(CR_e, (U.f_i : U_f, \emptyset)) \quad U, U_f \text{ are fresh}}{e.f_i : U_f \mid S_e \cup S_f \cup S' \mid R_e \mid CR \mid BR}$
GTC-INVK	$\frac{\begin{array}{l} e : T_e \mid S_e \mid R_e \mid CR_e \mid BR_e \\ CR_m = (U.m : \langle \overline{U}_y \triangleleft \overline{U}_p \rangle \overline{U} \rightarrow U', \emptyset) \quad \overline{e} : \overline{T} \mid \overline{S} \mid \overline{R} \mid \overline{CR} \mid \overline{BR} \\ \overline{S}_p = \{V <: \{U_y \mapsto V\} U_p\} \quad \overline{S}_s = \{S <: \{U_y \mapsto V\} U\} \\ \vdash \overline{V} \text{ ok} \mid S' \mid CR' \mid BR' \quad S_u = \{U = \{\overline{U}_y \mapsto \overline{V}\} U'\} \cup S_e \\ BR _{S_b} = \text{merge}_{BR}(BR_e, \overline{BR}, BR', (T_e \rightsquigarrow U)_{opt}) \end{array} \quad \overline{U}, U', \overline{U}, \overline{U}_p, \overline{U}_y \text{ are fresh}}{e.m \langle \overline{V} \rangle (\overline{e}) : U \mid S_u \cup \overline{S} \cup \overline{S}_p \cup \overline{S}_s \cup S_r \cup S_{cr} \cup S' \cup S_b \mid R \mid CR \mid BR}$
GTC-NEW	$\frac{\begin{array}{l} \vdash N \text{ ok} \mid S' \mid CR' \mid BR' \quad \overline{e} : \overline{T} \mid \overline{S} \mid \overline{R} \mid \overline{CR} \mid \overline{BR} \quad \overline{U} \text{ is fresh} \\ CR_f = (C.\text{init}(\overline{U}), \emptyset) \quad \overline{S}_s = \{T <: U\} \quad R _{S_r} = \text{merge}_R(R) \\ CR _{S_{cr}} = \text{merge}_{CR}(CR_f, \overline{CR}, CR') \quad BR _S = \text{merge}_{BR}(BR, \overline{BR}, BR') \end{array}}{\text{new } N(\overline{e}) : N \mid S' \cup \overline{S} \cup \overline{S}_s \cup S_r \cup S_{cr} \cup S \mid R \mid CR \mid BR}$
GTC-METHOD	$\frac{\begin{array}{l} e : T_e \mid S_e \mid R_e \mid CR_e \mid BR_e \quad \vdash \overline{T} \text{ ok} \mid S_t \mid CR_t \mid BR_t \\ \vdash T_0 \text{ ok} \mid S_z \mid CR_z \mid BR_z \quad \vdash \overline{P} \text{ ok} \mid S_p \mid CR_p \mid S_p \\ U_c, U_d, U \text{ are fresh} \quad \overline{S}_x = \{T = R_e(x) \mid x \in \text{dom}(R_e)\} \\ S_c = \{U_c = R_e(\text{this}) \mid \text{this} \in \text{dom}(R_e)\} \\ R_e - \text{this} - \overline{x} = \emptyset \quad BR _{S'} = \text{merge}_{BR}(BR_e, BR_t, BR_z, BR_p) \\ \text{removeBR}(\overline{Y} \triangleleft \overline{P}, BR) = BR' _{S'} \quad S_o = (T_0 <: U)_{opt} \\ CR _{S_{cr}} = \text{merge}_{CR}(CR_e, (U_c.\text{extends}: U_d, \emptyset), (U_d.m : \overline{T} \rightarrow U, \emptyset)_{opt}) \\ CR' _{S_r} = \text{merge}_{CR}(CR, CR_p, CR_z, CR_t) \\ S = S_e \cup S_t \cup S_z \cup S_p \cup S_o \cup S_c \cup S' \cup S_{cr} \cup \overline{S}_x \cup \{T_e <: T_0\} \end{array}}{\langle \overline{Y} \triangleleft \overline{P} \rangle T_0 \ m(\overline{T} \ \overline{x}) \ \{\text{return } e\} \text{ OK} \mid S \mid U_c \mid CR \mid BR'}$
GTC-CLASS	$\frac{\begin{array}{l} \vdash \overline{N} \text{ ok} \mid S_n \mid CR_n \mid BR_n \quad \vdash N \text{ ok} \mid S' \mid CR' \mid BR' \\ \vdash \overline{T} \text{ ok} \mid S_t \mid CR_t \mid BR_t \quad K = C(\overline{U} \ \overline{g}, \overline{T}' \ \overline{f}) \{\text{super}(\overline{g}); \text{this}.\overline{f} = \overline{f}\} \\ \overline{M} \text{ OK} \mid S \mid U \mid CR \mid BR \quad BR _S = \text{merge}_{BR}(\overline{BR}, BR_n, \overline{BR}', BR_t) \\ \text{removeBR}(\overline{X} \triangleleft \overline{N}, BR) = \emptyset _{S_r} \quad S_b = S_n \cup S' \cup S_t \cup S_r \cup \{U = C \langle \overline{X} \rangle\} \\ CR _{S_{cr}} = \text{merge}_{CR}((N.\text{init}(\overline{U}), \emptyset), \overline{CR}, CR_n, CR', CR_t) \end{array}}{\text{class } C \langle \overline{X} \triangleleft \overline{N} \rangle \triangleleft N \{\overline{T} \ \overline{f}; K \ \overline{M}\} \text{ OK} \mid S_b \cup \overline{S} \cup S \cup \overline{S}_{eq} \cup S_{cr} \mid CR}$

Figure 10.9.: A co-contextual formulation of the type system of FGJ.

the type T_e is a variable or a nonvariable type. Consequently, the type checker cannot decide whether to generate a bounded requirement or not. To solve this, we introduce the optional bounded requirements $(T_e \rightsquigarrow U)_{opt}$, where U a fresh unification variable. This special requirement is considered and has to be removed if T_e is a type variable. Otherwise, the requirement is invalidated and an quality constraint is generated indicating that $T_e = U$. The dual to looking up the field type is introducing a field requirement $(U.f : U_f, \emptyset)$, where U_f is a fresh unification variable, which is a placeholder for the actual field type.

The rule GTC-INVK is used to type check method invocation. The type checker starts with the expression e and the method arguments \bar{e} . The dual to looking up the bound of the type of expression e is generating an optional bounded requirement, as in the case of field accesses. The operation of looking up the method signature of m in CT is dual to generating a new requirement for the method m $(U.m \langle \overline{U}_y \triangleleft \overline{U}_p \rangle : \overline{U} \rightarrow U')$. This requirement has a fresh type variable U as receiver since the bound of T_e is not known. Also, the parameter and return types of the required method are unknown, therefore, we generate fresh unification variables, as placeholders for their actual types. We generate fresh unification variables for the generic parameters of the method (\overline{U}_y) and their bounds (\overline{U}_p) . The relation between them is added to the bounded requirements set. All subtype relations are recorded in the constraints S_p, S_s .

The rule GTC-NEW is used to type check the object creation in co-contextual FGJ. The dual of looking up the constructor signature is generating a constructor requirement $C.\text{init}(\overline{U})$. The actual constructor signature of the class C is not known because the class table is removed. Therefore, we generate fresh unification variables. The context, bounded and class table requirements are merged yielding resulting sets of requirements and constraints.

The rule GTC-METHOD is used to type check method declarations. We describe only the required changes done to the corresponding rule TC-METHOD in co-contextual FJ to support generic types. Initially, we consider the generic parameters of the declared method. Type variables \overline{Y} are part of the method declaration and we know their actual bounds \overline{P} . Therefore, the bounded requirements regarding \overline{Y} are now removed from the bounded requirements set. However, we do not know the bounds of the class U_c . The rest of the bounded requirements will be satisfied when we encounter the class corresponding to U_c . The most significant change is regarding method overriding and how it behaves in FGJ. Overriding a method in FGJ allows the methods to have covariant return types. Method overriding is already supported by the co-contextual FJ, but methods must have the same signature in class U_c and supper class U_d . To this end, we introduced optional method requirement. In co-contextual FJ this method requirement had the same signature with the receiver U_d , which is superclass of U_c , as the declared method m in U_c . This optional requirement looks like $(U_d.m : T \rightarrow U, \emptyset)_{opt}$. Since FGJ allows covariant return types, we have to make sure that the overriding of m is valid. The parameter types of the overriding and overridden methods are the same. Since the return types are covariant, we generate a fresh unification variable U as placeholder for the return type of m in U_d . Then, we generate a new constraint to relate the return type T_0 of m in U_c with the return type

U of m in U_d . This constraint is called the optional constraint $(T_0 <: U)_{opt}$. As for the optional requirement, this constraint is optional because we do not know whether there will be a declaration of m in U_d , or not. Therefore, this constraint is taken into consideration if there exists a declaration of m in the superclass U_d of U_c . Otherwise, the optional constraint is invalidated and not considered while solving the other constraints.

The rule for class declaration in co-contextual FGJ is the rule `GTC-CLASS`. This rule is almost the same as the corresponding rule `TC-CLASS` in co-contextual FJ. The only difference are the bounded requirements. Since the class C declares the bounds of its generic type variables, then the requirements on these type variables are satisfied and removed from the requirements set. As a result, the bounded requirements set is empty.

10.5. Summary

We have extended FJ with generics, a powerful language feature for parameterized classes and methods. Co-contextualizing FGJ was challenging because the generic types introduced completely new type structures, that required changes in the requirements, constraints.

The requirements were changed w.r.t. their operations of merge and remove. Two requirements can be merged if the receiver classes are the same and have equal generic parameters. Moreover, the remove operation has to distinguish the cases when the fields or methods have ground or generic types. We have introduced subtype substitution constraints to indicate subtype relations between types and the bounds of the generic type variables in the program.

In addition, the bounds of the generic types introduced a new requirement set for bounded requirements and accompanying operations for merge and remove. FGJ ensures that generic types are well-formed, using well-formedness judgement. Hence, by systematically applying dualism, we constructed the co-contextual well-formed rules.

Finally, the translation of the FGJ typing rules to the co-contextual typing rules was very involved because the premises include many additional constructs and operations. For example, in the rule `GTC-INVK` the lack of the class table and the context for type bounds requires fresh unification variable for the generic parameters and their bounds, and to keep track of their relations.

In conclusion, we have seen that the constraints are similar to the introduction of parametric polymorphism in PCF in Chapter 4. However, the bounded generic types introduced additional constructs. While the translation was very involved in terms of size of the premises and elements of the judgements that need to be considered, the actual translation from the contextual type systems was systematic and eased due to our dualism technique.

11. Co-Contextual Featherweight Java with Method Overloading

In this chapter, we extend the co-contextual FJ with method overloading (FMJ). Method overloading is a powerful language extension, because a method of the same name can have multiple implementations with varying signatures. This affects the class table requirements defined previously for FJ. That is, for method overloading a method requirement is satisfied when the type checker finds the most specific declaration of the required method. Hence, the type checker cannot satisfy requirements immediately when finding a fitting method, but has to resolve the best candidate for the required methods, which makes overloading a challenging feature to co-contextualize.

In the following, we describe the traditional FMJ, i.e., the types, syntax, and definition for minimal selection, and then provide the contextual typing rules. Next, we present the co-contextual structure of class table requirements for FMJ. Then, we extend the operations on the class table requirements to support method overloading and illustrate the remove operation with an example. Next, we construct the co-contextual typing rules for FMJ by applying our dualism technique. Finally, we discuss the impact of co-contextual FMJ on efficient incremental type checkers.

11.1. Featherweight Java with Method Overloading

In this section, we show the syntax, types and minimal selection definition of FMJ presented by Bettini et al [BCV09]. Then, we describe the FMJ typing rules.

11.1.1. Featherweight Java with Method Overloading: Types, Syntax and Minimal Selection

The types of FMJ are the types of FJ with *multi-types*, which are used to represent the multiple declarations of a method. Multi-types are a set of arrow types, each corresponding to signatures of a method and they have the form:

$$\{\overline{C}_1 \rightarrow C_1, \dots, \overline{C}_n \rightarrow C_n\}$$

The sequence notion is extended to multi-types to present them in a compact form $\{\overline{C} \rightarrow C\}$. Furthermore, the sequence notion is extended to support multi-method declarations:

$$\overline{C} \ m(\overline{C} \rightarrow C) \ \text{return } e;$$

We ensure that multi-types are well-formed given the definition below:

Definition 2 (Well-Formedness of Multi-Types). *A multi-type $\{\overline{T} \rightarrow T\}$ is well-formed, if $\forall (\overline{T}_i \rightarrow T_i), (\overline{T}_j \rightarrow T_j) \in \{\overline{T} \rightarrow T\}$:*

1. $\overline{T}_i \neq \overline{T}_j$
2. $\overline{T}_i <: \overline{T}_j \Rightarrow T_i <: T_j$

In FMJ, a method m can have multiple declarations corresponding to it. Hence, the return type of an invocation of the method m should be chosen among these multiple declarations. But, how to choose the best candidate? The solution is to select the most specific declaration, if possible. This is realized by applying minimal selection on the multi-type of an invoked method. In the following, we give the auxiliary functions needed to perform the minimal selection. Let us suppose that an invoked method m has parameter types \overline{C} and a receiver class C . Minimal selection for the method m is realized among the declarations of m in C and superclasses of C . Therefore, to perform minimal selection is required to have all method declarations in C and superclasses of it. Minimal selection operates only on the parameter types of a method and not on the return type. The set of methods from where we select the most specific declaration is defined by the match function, as shown below:

Definition 3 (Matching parameter types). *Given some parameter types \overline{C} and a multi-type $\{\overline{T} \rightarrow T\}$: $\text{match}(\overline{C}, \{\overline{T} \rightarrow T\}) = \{\overline{T}_i \rightarrow T_i \in \{\overline{T} \rightarrow T\} \mid \overline{C} <: \overline{T}_i\}$*

The resulting set of match contains methods having parameter types, which are supertype of \overline{C} .

Afterwards, we apply the function MIN on the resulting set of match to get a set of minimal arrow types, as shown below:

Definition 4 (Minimal parameter types). *Given a set of signatures $\{\overline{T} \rightarrow T\}$: $\text{MIN}(\{\overline{T} \rightarrow T\}) = \{\overline{T}_i \rightarrow T_i \in \{\overline{T} \rightarrow T\} \mid \forall (\overline{T}_j \rightarrow T_j) \in \{\overline{T} \rightarrow T\} \text{ s.t. } \overline{T}_i \neq \overline{T}_j, \overline{T}_j \not<: \overline{T}_i\}$*

Finally, we give the definition of the most specialized selection. To realize minimal selection, we use the above-defined auxiliary functions.

Definition 5 (Most specialized selection). *Given some parameter types \overline{C} and a multi-type $\{\overline{B} \rightarrow B\}$, then*

$\text{minsel}(\overline{C}, \{\overline{B} \rightarrow B\}) = \overline{B}_i \rightarrow B_i$ if and only if
 $\text{MIN}(\text{match}(\overline{C}, \{\overline{B} \rightarrow B\})) = \{\overline{B}_i \rightarrow B_i\}$

Minimal selection on a multi-type is considered to be successful if the resulting set after applying MIN has a single element, which is the most specialized selection. Otherwise, minimal selection is undefined. That is, we cannot deduce a minimal type for the invoked method because of the ambiguities on the parameter types.

Let us illustrate the minimal selection given the example below:

```

class A extends Object
class D extends Object
class B extends A
class C extends B

```

```

class E extends Object{
  Pair(B, B) m(B x) { ... }
}
class F extends E{
  Pair(D, D) m(D x) { ... }
  Pair(A, A) m(A x) { ... }
}

```

```
(new F()).m(new C())
```

We have a method invocation for m with argument type C and receiver class F . Initially, we look for declarations of m in F and superclasses of F . We find three declarations, which are $\{B \rightarrow \text{Pair}(B, B), D \rightarrow \text{Pair}(D, D), A \rightarrow \text{Pair}(A, A)\}$. Then, we apply the function *match* to select the method declarations that have parameter types, which are supertype of C i.e., $\text{match}(C, \{B \rightarrow \text{Pair}(B, B), D \rightarrow \text{Pair}(D, D), A \rightarrow \text{Pair}(A, A)\}) = \{B \rightarrow \text{Pair}(B, B), A \rightarrow \text{Pair}(A, A)\}$. Finally, we apply *MIN*, i.e., $\text{MIN}(\{B \rightarrow \text{Pair}(B, B), A \rightarrow \text{Pair}(A, A)\}) = \{B \rightarrow \text{Pair}(B, B)\}$, and the result is a set with a single element, indicating that the minimal selection is successful. As a result, the return type of the invoked m is $\text{Pair}(B, B)$.

Let us change the example above to illustrate method ambiguities.

```

class A extends Object
class C extends Object
class B extends A
class D extends C

```

```

class F extends Object{
  Pair(B, C) m(B x, C y) { ... }
  Pair(A, D) m(A x, D y) { ... }
}

```

```
(new F()).m(new B(), new D())
```

Argument types of the invoked method m are (B, D) . Both declarations of m in F have parameter types, which are supertypes of (B, D) . Applying *MIN* results in a set of two elements and the most specific signature cannot be decided, i.e., $\text{MIN}(\{(B, C) \rightarrow \text{Pair}(B, C), (A, D) \rightarrow \text{Pair}(A, D)\}) = \{(B, C) \rightarrow \text{Pair}(B, C), (A, D) \rightarrow \text{Pair}(A, D)\}$ because parameter types (B, C) and (A, D) are not subtype of each other, i.e., $B <: A, C \not<: D$.

11.1.2. Featherweight Java with Method Overloading Typing Rules

In this section, we present the FMJ typing rules. We show only the FJ typing rules that change to support method overloading, which are T-INVK, T-METHOD and T-CLASS, as shown

11. Co-Contextual Featherweight Java with Method Overloading

in Figure 11.1.

MT-INVK	$\frac{\Gamma; CT \vdash e : C_e \quad \Gamma; CT \vdash \bar{e} : \bar{C} \quad \text{mtypesel}(m, C_e, \bar{C}, CT) = \bar{D} \rightarrow D}{\Gamma; CT \vdash e.m(\bar{e}) : D}$
MT-METHOD	$\frac{\bar{x} : \bar{C}; \mathbf{this} : C; CT \vdash e : E_0 \quad E_0 <: C_0 \quad \text{class } C \text{ extends } D \{ \bar{C} \bar{f}; K; \bar{M} \}}{C; CT \vdash C_0 m(\bar{C} \bar{x}) \{ \mathbf{return } e \} \text{ OK}}$
MT-CLASS	$\frac{\begin{array}{l} K = C(\bar{D}' \bar{g}, \bar{C}' \bar{f}) \{ \mathbf{super}(\bar{g}); \mathbf{this}.\bar{f} = \bar{f} \} \\ \text{fields}(D, CT) = D.\mathbf{init}(\bar{D}') \quad C; CT \vdash \bar{M} \text{ OK} \\ \text{mtype}(m, C) = \{ \bar{B} \rightarrow B \} \wedge \text{well-formed}(\{ \bar{B} \rightarrow B \}) \forall m \in \bar{M} \end{array}}{CT \vdash \mathbf{class } C \text{ extends } D \{ \bar{C} \bar{f}; K \bar{M} \} \text{ OK}}$

Figure 11.1.: Typing Rules of Featherweight Java with Method Overloading.

First, we consider the rule for method invocations MT-INVK. The type checker starts type checking the receiver expression e and the arguments of the invoked method. In FJ, the type checker looks up the signature of m in the class table CT , which is declared in C_e or superclasses of it. However, in FMJ, the invoked method m can have multiple declarations. Therefore, we have to collect all declarations of the method m and select the most specific sinature by applying minsel, as described previously. To do so, FMJ uses the auxiliary function mtypesel, which is shown in Figure 11.2. Moreover, the subtype constraint, which ensures that the argument types are subtype of the parameter types, is not added. This check is ensured while selecting the matching declarations to the invoked method using the function match.

$\frac{\text{mtype}(m, C, CT) = \{ \bar{B} \rightarrow B \} \quad \text{minsel}(\bar{C}, \{ \bar{B} \rightarrow B \}) = \bar{D} \rightarrow D}{\text{mtypesel}(m, C, \bar{C}, CT) = \bar{D} \rightarrow D}$

Figure 11.2.: Method type lookup for method overloading.

The rule MT-METHOD is used to type check method declarations. The premises of this typing rule are almost the same as the premises of the corresponding rule in FJ, except the check for method overriding. This check is not necessary anymore because any new signature regarding method m is legal. If the same signature is already defined for m in a

superclass of C , then the current signature $\overline{C} \rightarrow C_0$ overrides the existing one, otherwise, it is considered as a different signature for m .

Finally, we describe the rule MT-CLASS for class declarations. This typing rule has almost the same premises as the corresponding typing rule in FJ, with an additional check for the multi-types well-formedness. All multi-types of all declared methods in C should be well-formed, which is ensured by using the well-formed function from Definition 2.

11.2. Co-Contextual Structures for Featherweight Java with Method Overloading

In this section, we describe systematic changes to co-contextual FJ in order to support method overloading.

11.2.1. Co-contextual Constraint

The set of constraints used in co-contextual FJ is extended with a new constraint to support overloading, which is shown below:

$$s ::= \dots \mid \overline{U} \rightarrow U = \text{minselCo}(T, \overline{T}, \text{mltype}) \quad \text{minimal selection constraint}$$

The minimal selection constraint is introduced to compute the most specific signature of a required method. We will discuss this constraint in more detail, when describing the remove operation on method requirements.

However, the types of co-contextual FMJ include multi-types in addition to the types described for co-contextual FJ. Multi-types are the same as for FMJ, described in the previous section.

We use dualism to translate FMJ to co-contextual FMJ. As for FJ, contexts and class tables are replaced by the dual structures of context and class table requirements. The operations on the requirements are dual to the operations on the contexts and class tables. Class table requirements and their operations change to support multiple declarations of a method.

11.2.2. Structure of Class Table Requirements

Method overloading allows methods to have different declarations and introduces multi-types to represent these declarations. This affects the results of method invocations. Consequently, the method requirements and the operations of merge and remove on method requirements are affected.

In the following, we describe the new structure of method requirements. In FJ, a method requirement consists on an unknown receiver class, the required method, its signature, and a condition. This requirement is satisfied once the type checker encounters a method clause in the class table that has the same name as the required method. However, in FMJ, a method requirement cannot be satisfied given only one method declaration in the class table because a method can correspond to several declarations and the return type of

the required method is obtained from the most specific signature. Therefore, we need to keep track of all method declarations that have the same name as the required method, in order to apply minimal selection to them. To enable this, we add a structure to the method requirements, which contains the information about all declarations corresponding to the required method. In addition to the method declarations, we need to keep track of the classes they belong to. The reason is that to perform minimal selection, we should consider only the method declarations that belong to the receiver class of the required method or superclasses of it, which we do not know because of nominal typing. A method requirement, which supports method overloading looks like:

$$CR_m = (T_c.m : \overline{U} \rightarrow U, \overline{T}, mltype, cond).$$

To each method requirement, we assign fresh unification variables as its signature. That is, \overline{U} and U are fresh, which are placeholders for the signature deduced from the minimal selection applied to all declarations of the method m . \overline{T} are the argument types of the invoked method, which are considered while performing the match function. $mltype$ is a map from classes to sets of signatures, which has the form $mltype \subset C \times \{\overline{C} \rightarrow C'\}$. A set of signatures is obtained from the declarations of all methods within a class that have a name matching the required method.

11.3. Operations on Class Table Requirements

In the following, we describe the merge and remove operations on method requirements in the presence of overloading.

11.3.1. Merge Operation

The merge operations is realized on methods that have the same name. However, in FMJ, methods can have multiple declarations and merging two method requirements is more complex than in the case of FJ. In FMJ, two method requirements are merged if they have the same name, receiver classes and argument types. We include the comparison of the argument types in the decision for merging because they are a decisive element when realizing minimal selection. The function `match` selects only signatures, which have parameter types that are supertypes of the argument types. If the argument types are different, `match` would result in different sets, giving different minimal selections.

Because of overloading the same method name can have arguments that vary in arity and types. Let us first consider the length of the arguments. We use the auxiliary function `lng` to get the length of a sequence of types. If the lengths of the arguments are the same than the two requirements are potential candidates for merging, otherwise not, as shown in Equation 11.1. The results of the merge in both cases is shown in Figure 11.3.

$$\text{merge}(CR_1, CR_2) = \begin{cases} CR'_m|_{S'_m} & \text{if } \text{lng}(\overline{T}_1) \neq \text{lng}(\overline{T}_2) \\ CR_m|_{S_m} & \text{if } \overline{T} = \overline{T}' \end{cases} \quad (11.1)$$

$$\begin{aligned}
 CR'_m &= \{(T_1.m : \overline{U}_1 \rightarrow U_1, \overline{T}_1, mltype_1, cond_1) \\
 &\quad \cup (T_2.m : \overline{U}_2 \rightarrow U_2, \overline{T}_2, mltype_2, cond_2)\} \\
 S'_m &= \emptyset \\
 CR_m &= \{(T_1.m : \overline{U}_1 \rightarrow U_1, \overline{T}_1, mltype_1, cond_1 \cup (T_1 \neq T_2) \cup (\overline{T}_1 \neq \overline{T}_2)) \\
 &\quad \cup (T_2.m : \overline{U}_2 \rightarrow U_2, \overline{T}_2, mltype_2, cond_2 \cup (T_1 \neq T_2) \cup (\overline{T}_1 \neq \overline{T}_2)) \\
 &\quad \cup (T_1.m : \overline{U}_1 \rightarrow U_1, \overline{T}_1, mltype_1, cond_1 \cup (T_1 = T_2) \cup (\overline{T}_1 = \overline{T}_2)) \\
 &\quad | (T_1.m : \overline{U}_1 \rightarrow U_1, \overline{T}_1, mltype_1, cond_1) \in CR_1 \wedge \\
 &\quad (T_2.m : \overline{U}_2 \rightarrow U_2, \overline{T}_2, mltype_2, cond_2) \in CR_2\} \\
 S_m &= (U_1 = U_2 \text{ if } (T_1 = T_2) \cup (\overline{T}_1 = \overline{T}_2)) \cup (\overline{U}_1 = \overline{U}_2 \text{ if } (T_1 = T_2) \cup (\overline{T}_1 = \overline{T}_2)) \\
 &\quad | (T_1.m : \overline{U}_1 \rightarrow U_1, \overline{T}_1, mltype_1, cond_1) \in CR_1 \wedge \\
 &\quad (T_2.m : \overline{U}_2 \rightarrow U_2, \overline{T}_2, mltype_2, cond_2) \in CR_2\}
 \end{aligned}$$

 Figure 11.3.: Merge operation of method requirements CR_1 and CR_2 .

Next, we have to consider the argument types. Because of nominal typing, we do not know the actual argument types of the required method m . Therefore, we cannot compare the argument types, to decide whether the requirements can be merged or not. Therefore, we add type equalities and inequalities to the conditions to indicate that two requirements will be overlapping or not, i.e., $\overline{T}_1 = \overline{T}_2$ and $\overline{T}_1 \neq \overline{T}_2$, respectively. Also, we add equalities and inequalities for the receiver classes, as in the case of co-contextual FJ.

In addition, the merge operation generates the conditional constraints S_m , which indicates that the most specific parameter and return types of the two method requirements are the same if the receiver classes and the argument types are the same.

11.3.2. Remove Operation and Minimal Selection of the Required Method

In the following, we describe the removal of method requirements, as shown in Figure 11.4.

A method requirement is generated at method invocation. In FMJ, the return type of an invoked method is part of the most specific signature deduced for that method. Therefore, a method requirement is satisfied when the most specific signature is computed given all declarations corresponding to the required method. To deduce the most specific signature, a method requirement cannot be satisfied until all method declarations in all class hierarchies that have the same name as the required method, are provided. As a result, requirements regarding method m cannot be removed from the requirements set even when we know the actual receiver classes because at any point in time could be

$$\begin{aligned}
\text{addMs}(C, \overline{M}, CT) &= \overline{C.m : \overline{C} \rightarrow C'} \cup CT \\
\\
\text{removeM}(C, C' \ m(\overline{C} \ \overline{x}) \ \{\mathbf{return} \ e\}, CR) &= CR' \\
\text{where } CR' &= \{(T.m : \overline{U} \rightarrow U, \overline{T}, \text{mltype.addC}(C \rightarrow \{\overline{C} \rightarrow C'\}), \text{cond} \wedge (T \neq C)) \\
&\quad | (T.m : \overline{U} \rightarrow U, \overline{T}, \text{mltype}, \text{cond}) \in CR\} \\
&\quad \cup (CR \setminus \overline{(T.m : \overline{U} \rightarrow U, \overline{T}, \text{mltype}, \text{cond})}) \\
\text{removeMs}(C, \overline{M}, CR) &= CR' \\
\text{where } CR' &= \{CR_m \mid (C' \ m(\overline{C} \ \overline{x}) \ \{\mathbf{return} \ e\}) \in \overline{M} \text{ and} \\
&\quad \text{removeM}(C, C' \ m(\overline{C} \ \overline{x}) \ \{\mathbf{return} \ e\}, CR) = CR_m\}
\end{aligned}$$

Figure 11.4.: Add and remove operations of method clauses.

encountered a declaration of m , which alters the result of the minimal selection. This hampers the gradual discharge of method requirements.

The removal of a method requirement m , given a declaration of m , consists on updating the condition and the *mltype*. The signature of the declared method is added to *mltype* with the corresponding class C . In order to add elements to the map *mltype*, we use the auxiliary function *addC*. For each $(C \rightarrow \{\overline{C} \rightarrow C'\})$, it is checked if it exists an entry for C in *mltype* or not. If there is an entry for C , then it is updated only the corresponding multi-type with $\overline{C} \rightarrow C'$. Otherwise, we add a new entry for C . The condition is updated with the type inequality $T \neq C$, as in the case of co-contextual FJ.

However, *remove* in the presence of method overloading cannot satisfy requirements because the type of the required method is not decided given a single method declaration. Hence, a conditional equality constraint is not generated.

At the end of type checking, when all class declarations are encountered, we still have remaining requirements, which are method requirements. Therefore, we add a new operation on the method requirements, which we call *discharge*.

$$\text{discharge}(\overline{(T.m : \overline{U} \rightarrow U, \overline{T}, \text{mltype}, \text{cond})}) = CR|_S$$

The result of *discharge* is new set of requirements and a set of constraints. CR contains the remaining method requirements that cannot be discharged from the requirement set. The discharged method requirements are those with unsatisfiable conditions. Any remaining requirement in CR would indicate that the program is not well-typed because the resulting set of requirements after encountering all its class declarations is not empty.

The set of constraints S are minimal selection constraints. These constraints assign to the required methods the most specific signatures, which are the results of minimal selections.

$$\overline{U} \rightarrow U = \text{minselCo}(T, \overline{T}, \text{mltype})$$

To compute the minimal selection in a co-contextual setting, we use the auxiliary

function $\text{minselCo}(T, \overline{T}, \text{mltype})$. Note that the minimal selection can be solved, when the type checker knows the actual type C of the receiver class T and the actual argument types \overline{C} of \overline{T} . In contrast to minsel , minselCo has an additional field for the receiver class C of the required method. The reason is to select the method signatures which are declared in C or superclasses of C . Hence, we have to narrow down the map mltype to a multi-type, before we apply match shown in Definition 3, as shown below:

Definition 6 (Multi-type from matching classes). *Given a class C and a set of declarations mltype of a method m , where $\text{mltype} = \{D \rightarrow \{\overline{C} \rightarrow \overline{C'}\}\}$ then the multi-type of m corresponding to the hierarchy of C is:*

$$\text{matchSup}(C, \text{mltype}) = \bigcup (\{\overline{C_i} \rightarrow \overline{C'_i}\} \mid (D_i \rightarrow \{\overline{C_i} \rightarrow \overline{C'_i}\}) \in \text{mltype} \wedge C <: D_i)$$

Applying matchSup to mltype is a preliminary step we have to realize before matching parameter types. As a result, the definition of minselCo is shown below:

Definition 7 (Co-contextual most specialized selection.). *Given a class C , parameter types \overline{C} , and a $\text{mltype} \{D \rightarrow \{\overline{C} \rightarrow \overline{C'}\}\}$, then*

$\text{minselCo}(C, \overline{C}, \text{mltype}) = \overline{B_i} \rightarrow B_i$ if and only if

$$\text{MIN}(\text{match}(\overline{C}, \text{matchSup}(C, \text{mltype}))) = \{\overline{B_i} \rightarrow B_i\}$$

The result of minselCo has three possible alternatives. First, the resulting set is empty, which means that no declaration for the required method m was found in the receiver class or its superclasses. Therefore, the respective requirement is not satisfied and the program is not well-typed. Second, the result of minselCo is a set with only one element. In this case, declarations for m were found, the respective requirement is satisfied and the minimal selection was successful because the resulting set has a single element. Third, the result of minselCo is a set with more than one element. Then, the constraint is not solvable since we cannot decide the most specialized type, which indicates possible ambiguities. Therefore, the program is not well-typed.

11.3.3. Remove Operation with Overloading by Example

We describe the remove operation step by step and consider the example given in Section 11.1.

```

class A extends Object
class D extends Object
class B extends A
class C extends B

class E extends Object{
  Pair(B, B) m (B x)
}
class F extends E{
  Pair(D, D) m(D x)
  Pair(A, A) m(A x)
}
    
```

$\text{genF}().m(\text{new } C(\dots))$

Suppose $\text{genF}()$ will return an F instance.

Method invocation $\text{genF}().m(\text{new } C(\dots))$ will generate the method requirement $CR_m = U.m : U_1 \rightarrow U_2, C, \emptyset, \emptyset$. We apply remove on CR_m , given the different declarations of m . In the next steps, class declarations E and F are type-checked.

First, the type checker encounters the method declarations of m in F . Adding $F.m : D \rightarrow \text{Pair}(D, D)$ and $F.m : A \rightarrow \text{Pair}(A, A)$ to the class table is dual to removing the corresponding method requirements $CR'_m = \text{removeMs}(U.m : U_1 \rightarrow U_2, C, \emptyset, \emptyset, \overline{M})$, which gives the requirement $CR'_m = (U.m : U_1 \rightarrow U_2, C, (F \rightarrow \{D \rightarrow \text{Pair}(D, D), A \rightarrow \text{Pair}(A, A)\}), U \neq F)$.

Second, we consider the declaration of m in E and the result of remove is $CR = (U.m : U_1 \rightarrow U_2, C, (F \rightarrow \{D \rightarrow \text{Pair}(D, D), A \rightarrow \text{Pair}(A, A)\}), E \rightarrow \{B \rightarrow \text{Pair}(B, B)\}), U \neq F, U = F, U \neq E)$.

In the next step, the type checker processes all the other class declarations A, B, C, D , but they do not have any declaration regarding the method m . We suppose the type checker encounters a declaration for getF , which has the return type F and does not affect the requirement set. Given this information, the type checker deduces the actual type of the receiver class of the required method, i.e., $U = F$.

Finally, at this point of type checking, all class declarations of the given program are type checked. Hence, we apply discharge and a minimal selection constraint is generated. Considering that $U = F$, we get $(U_1 \rightarrow U_2) = \text{minselCo}(U, C, \text{mltype}) = \text{minselCo}(F, C, F \rightarrow \{D \rightarrow \text{Pair}(D, D), A \rightarrow \text{Pair}(A, A)\}, E \rightarrow \{B \rightarrow \text{Pair}(B, B)\})$. The result of computing the minimal selection is a single element $\{B \rightarrow \text{Pair}(B, B)\}$. Therefore, the actual most specific type of the required method is $U_1 \rightarrow U_2 = B \rightarrow \text{Pair}(B, B)$; the return type of the invoked method m is $\text{Pair}(B, B)$. Moreover, the resulting set from discharge is empty because $U \neq F$ does not hold, which makes the condition of the requirement CR unsatisfiable.

To conclude, the program is well-typed because the requirements set is empty and all constraints are solved.

11.4. Co-Contextual FJ with Method Overloading Typing Rules.

In the following, we describe the FMJ typing rules. Figure 11.5 shows the typing rules for method invocation, method declaration and class declaration.

Method invocation typing The rule MTC-INVK is used to type check method invocation, which is almost the same as the corresponding rule TC-INVK in co-contextual FJ. The two differences are: 1) we generate a new method requirement, which features method overloading and 2) we do not add the constraint $\overline{T} <: \overline{U}$. The fresh method requirement with overloading has the map of multi-types and the condition empty. The subtype constraint is not generated because $(\overline{U} \rightarrow U)$ represents the result of the minimal selection

$$\begin{array}{c}
\text{MTC-INVK} \frac{
\begin{array}{c}
e : T_e \mid S_e \mid R_e \mid CR_e \quad \bar{e} : \bar{T} \mid \bar{S} \mid \bar{R} \mid \bar{CR} \quad U, \bar{U} \text{ are fresh} \\
CR_m = (T_e.m : \bar{U} \rightarrow U, \bar{T}, \emptyset, \emptyset) \quad \text{merge}_R(R_e, R_1, \dots, R_n) = R_{|S_r} \\
\text{merge}_{CR}(CR_e, CR_1, \dots, CR_n) = CR'_{|S_{cr}}
\end{array}
}{e.m(\bar{e}) : U \mid \bar{S} \cup S_e \cup S_r \cup S_{cr} \mid R \mid CR}
\\[20pt]
\text{MTC-METHOD} \frac{
\begin{array}{c}
e_0 : T_e \mid S_e \mid R_e \mid CR_e \quad U_c, U_d \text{ are fresh} \\
S = \{T_e <: C_0\} \cup \{U_c = R_e(\text{this}) \mid \text{this} \in \text{dom}(R_e)\} \\
S_x = \{C = R_e(x) \mid x \in \text{dom}(R_e)\} \quad R_e - \text{this} - \bar{x} = \emptyset \\
CR_{S_{cr}} = \text{merge}_{CR}(CR_e, (U_c.\text{extends} : U_d, \emptyset))
\end{array}
}{C_0 \ m(\bar{C} \ \bar{x}) \ \{\text{return } e_0\} \ \text{OK} \mid S_e \cup S \cup \bar{S}_x \cup S_{cr} \mid U_c \mid CR}
\\[20pt]
\text{MTC-CLASS} \frac{
\begin{array}{c}
K = C(\bar{D}' \ \bar{g}, \bar{C}' \ \bar{f})\{\text{super}(\bar{g}); \text{this}.\bar{f} = \bar{f}\} \quad \bar{M} \ \text{OK} \mid S \mid U \mid CR \\
CR'_{|S_{cr}} = \text{merge}_{CR}((D.\text{init}(\bar{D}'), \emptyset), \bar{C}\bar{R}) \quad \bar{S}_{eq} = \{U = C\} \\
\text{mltypeM}(m, \bar{M}) = \{\bar{T} \rightarrow T\} \wedge \text{well-formed}(\{\bar{T} \rightarrow T\}) \ \forall m \in \bar{M}
\end{array}
}{\text{class } C \ \text{extends } D\{\bar{C} \ \bar{f}; \ K \ \bar{M}\} \ \text{OK} \mid \bar{S} \cup \bar{S}_{eq} \cup S_{cr} \mid CR'}
\end{array}$$

Figure 11.5.: A co-contextual formulation of the type system of FMJ.

for a method m and the function match, which is part of the minimal selection, ensures the subtype relation between parameter types.

Method typing The rule **MTC-METHOD** is dual to the method declaration rule in FMJ, where a method overriding check is not necessary. Hence, the co-contextual type checker does not generate an optional method requirement. However, a requirement for **extends** is generated to keep track of the relation between the class U_c and its superclass U_d , which is dual to looking up the superclass of C in the contextual rule **MT-METHOD**.

Class typing The typing rule **MTC-CLASS** is dual to the class declaration typing rule in FMJ. This typing rule is almost the same as the corresponding rule **TC-CLASS** in co-contextual FJ. The only difference is the well-formedness check for the multi-types of the methods declared in the class C . This check is ensured using the function well-formed, as in the rule **MT-CLASS** in contextual FMJ.

11.5. Impact of Co-contextual FJ with Method Overloading on Efficient Incremental Type Checkers

In contrast to the previously considered language features, method overloading has a severe impact on efficient incremental type checkers. The semantics of the overloading requires the type checker to resolve the most specialized signature of an invoked method

via a minimal selection. This operation involves a large amount of information that is not locally provided, which limits the gradual satisfaction of method requirements.

First, we need to know all method declarations corresponding to a required method. These declarations can be in the receiver class itself, but also in any of the superclasses. Second, we have to know all supertype relations of the arguments of the required method in order to find the most specific match. Therefore the type checker has to encounter a large number of complete class hierarchies. As a result, most of the computations of satisfying requirements and solving minimal selection constraints are deferred until the very end of type checking. Third, we need to recompute the minimal selection function again when a method changes.

Intuitively the previous incremental co-contextual type checkers have utilized the fact, that method requirements can be satisfied when any matching declaration is found in a program. Therefore, few constraints and requirements are deferred up to the root of the syntax tree. The respective optimizations of using memoization, incremental and continuous constraint solving, and in-depth merging of requirements were based on this property. As we have shown in the previous performance evaluations for incremental co-contextual PCF and FJ this was sufficient so far. In contrast, the method overloading would satisfy a method requirement too early if waiting only for any matching declaration, i.e., before knowing the actual result of the minimal selection and whether the minimal selection constraint is solvable or not. Hence, method requirements can be satisfied only at the end of type checking because a method requirement records the information to compute the minimal selection.

In conclusion, the formalism presented in this chapter is a faithful translation of method overloading, but needs additional refinement to be applicable to an efficient incremental co-contextual type checker. Formalizing the method overloading into a co-contextual type system has helped us in identifying the impact of the language feature in a systematic manner. We can now reason over specific properties needed for an efficient incremental type checker. First, we need to introduce a notion of the completion of a (super) class hierarchy, i.e., we can determine that the root class *Object* is reached and, hence, no further superclasses will be evident. With this notion the co-contextual formalism can satisfy requirements when all relevant class hierarchies have been traversed, i.e., for the receiver type and for the parameter types. Second, we need to introduce constraints for methods that participate in a minimal selection and faithfully translate the minimal selection semantics to these constraints.

We have further analyzed the implications of the method overloading construct by itself in an efficient incremental program analysis, in the context of IncA. IncA is a DSL for the definition and efficient evaluation of incremental program analysis in IDEs [SEV16]. IncA allows to easily build new relations and precisely model the dependencies between them. We were successful and IncA can efficiently maintain the relations required for method overloading when applying a delta to the program. IncA delivers incremental update times in the range of milliseconds, thereby showing that this most important performance characteristic is achievable.

12. Related Work

In this chapter, we discuss work, especially on OO languages, related to our co-contextual formulation of type rules and work related to incremental type checking.

Introducing type variables as placeholders for the actual types of variables, classes, fields, methods is a known technique in type inference [Pie02, PT00]. The difference is that we introduce a fresh class variable for each occurrence of a method m or fields in different branches of the typing derivation. Since fresh class variables are generated independently, no coordination is required while moving up the derivation tree, ensuring context and class table independence. Type inference uses the context to coordinate type checking of m in different branches, by using the same type variable. In contrast to type inference where context and class table are available, we remove them (no actual context and class table). Hence, in type inference inheritance relation between classes and members of the classes are given, whereas in co-contextual FJ we establish these relations through requirements. That is, classes are required to have certain members with unknown types and unknown inheritance relation, dictated from the surrounding program.

Also, in contrast to bidirectional type checking [Chr13, DK13] that uses two sets of typing rules one for inference and one for checking, we use one set of co-contextual type rules, and the direction of type checking is all oriented bottom-up; types and requirements flow up. As in type inference, bidirectional type checking uses context to look up variables. Whereas co-contextual FJ has no context or class table, it uses requirements as a structure to record the required information on fields, methods, such that it enables resolving class variables of the required fields, methods to their actual types.

Co-contextual formulation of type rules for FJ is related to the work on principal typing [Jim96, Wel02], and especially to principal typing for Java-like languages [AZ04]. A principal typing [AZ04] of each fragment (e.g., modules, packages, or classes) is associated with a set of type constraints on classes, which represents all other possible typings and can be used to check compatibility in all possible contexts. That is, principle typing finds the strongest type of a source fragment in the weakest context. This is used for type inference and separate compilation in FJ. They can deduce exact type constraints using a type inference algorithm. We generalize this and do not only infer requirements on classes but also on method parameters and the current class. Moreover, we developed a duality relation between the class table and class requirements that enables the systematic development of co-contextual type systems for OO languages beyond FJ.

The work by Ancona et al. [ALZ02a, ALZ02b], related to our co-contextual FJ, presents a framework to enable separate compilation of Java classes. These classes are considered as fragments in the framework. The work on separate compilation requires information on the superclass of the declared class to ensure that method overriding is valid. Also, at

12. Related Work

method invocation the receiver class of the invoked method and its superclass are known, so that they can construct a set of *selectable methods*. The aim is to perform separate compilation under a set of minimal type assumptions. They maintain two environments, local environment corresponding to the assumptions of the class under scrutiny and global environment keeps track of its superclass. There are two phases of type checking *intra – checking* that checks the program fragments and the *inter – checking* tries to satisfy the assumption on the fragments. On the other hand, in our co-contextual setting, we can type check method invocations, method declarations, or class declaration with no information on the receiver classes, that the declared methods are member of, or superclasses of them. Moreover, in our setting we can consider methods as fragments.

Related to our co-contextual FJ are the formulations used in the context of compositional compilation [ADDZ05] (continuation of the work on principal typing [AZ04]) and the compositional explanation of type errors [Chi01]. Ancona et al. [ADDZ05] type system partially eliminates the class table, namely only inside a fragment, and does not eliminate the context. Hence, type checking of parameters and **this** is coordinated and subexpressions are coupled through dependencies on the usage of context. In our technique, we eliminate both class table (not only partially) and context, therefore all dependencies are removed. By doing so we can enable compositional compilation for individual methods. To resolve the type constraints on classes, compositional compilation [AZ04] needs a linker in addition to an inference algorithm (to deduce exact type constraints), whereas, we use a constraint system and requirements. We use duality to derive a co-contextual type system for FJ and we also ensure that both formulations are equivalent (9.3). That is, we ensure that an expression, method, class, or program is well-typed in FJ if and only if it is well-typed in co-contextual FJ, and that all requirements are fulfilled. In contrast, compositional compilation rules do not check whether the inferred collection of constraints on classes is satisfiable; they actually allow to derive judgments for any fragment, even for those that are not statically correct.

Refactoring for generalization using type constraint is a technique Tip et al. [TKB03, TFK⁺11] used to manipulate types and class hierarchies to enable refactoring. Their technique uses variable type constraints as placeholders for changeable declarations. They use the constraints to restrict when a refactoring can be performed. Tip et al. are interested to find a way to represent the actual class hierarchy and to use constraints to have a safe refactoring and a well-typed program after refactoring. The constraint system used by Tip et al. is specialized to refactoring, because different variable constraints and solving techniques are needed. In contrast to their works, we use unification variables as placeholders for the actual types of required extends, constructors, fields, and methods of a class, in lack of the class table. We want to gradually learn the information about the class hierarchy. We are interested in the type checking technique and how to co-contextualize it and use constraints for type refinement.

Adapton [HPHF14] is a programming language where the runtime system traces memory reads and writes and selectively replays dependent computations when a memory change occurs. In principle, this can be used to define an “incremental” contextual type checker. However, due to the top-down threading of the context, most of the computation will be sensitive to context changes and will have to be replayed, thus yielding unsatisfactory

incremental performance. Given a co-contextual formulation as developed in this thesis, it might be possible to define an efficient implementation in Adapton.

The works on smart/est recompilation [SA93, Tic86] have a different purpose from ours, namely to achieve separate compilation they need algorithms for the inference and also the linking phase which are specific to SML. In contrast, we use duality as a guiding principle to enable the translation from FJ to co-contextual FJ. This technique allows us to do perform a systematic (but yet not mechanical) translation from a given type system to the co-contextual one. Our type system facilitates incremental type checking because we decouple the dependencies between subexpressions and the smallest unit of compilation is any node in the syntax tree. Moreover, we have investigated optimizations for facilitating the early solving of requirements and constraints.

The work by Shankar et al. [SB07] contributes on building an automatic incrementalizer for a class of data structure invariant checks that are written in an OO language, like Java or $C\sharp$. They looked into the work of Acar et al [ABH02] on self-adjusting computations that we described in Part II (Chapter 6) and wanted to investigate the automation of that style of incrementalization. They use some optimistic memoization algorithms, a technique that aggressively enables local computations to reconstruct a global result. They focus on the dynamic checks and incrementalize the dynamic-data structure invariant checks. On the other hand, the primer focus of this thesis is to build co-contextual type checkers, as a new way of type checking, then we show how to build fine grained incremental co-contextual type checkers. Moreover, co-contextual type checkers could be used not only for incrementalization but also for parallelization.

13. Part Summary

In this part, we have described our technique to construct co-contextual type checkers for OO languages, which are more challenging to build than the type checkers presented in the previous Part II. The reason is that the key features of OO languages: *subtype polymorphism*, *nominal typing* and *implementation inheritance*, dramatically increase the complexity of type systems. OO languages require class tables as an additional construct to the typing context to support these key features. To fully remove dependencies between subexpression of a program, we have removed the class tables. By systematically applying the dualism technique, we have introduced the dual structure of *class table requirements*, which are propagated bottom-up.

To support the inheritance and nominal typing of OO languages, we have introduced conditions (*cond*) as part of the requirements and conditional constraints. We have illustrated how conditions facilitate co-contextual type checking in the absence of the contextual class table that contains the fully-known class hierarchy. We have incorporated conditions into the co-contextual formalism, with the intuition that a class table requirement is discharged from the requirements set if at least one of the type equalities or inequalities in *cond* does not hold. This is important because a program in the co-contextual setting is well-typed if the requirements sets are empty.

To support subtype polymorphism, we have introduced an additional kind of class table requirement, the *optional method requirement*. This requirement ensures the consistency for method overriding, i.e., that a method *m* declared in a class *C* is consistent with the declaration (if there exists any) in the superclass of *C*. The novelty of the optional requirement is that if there is no declaration of *m* in the superclasses of *C*, this requirement is removed from the requirements set.

We have introduced operations on the class table requirements that are dual to the operations on class tables. In addition, we gave a detailed discussion on the operations for merging and removing class table requirements. We have described the usage and disused the necessity for conditions in the co-contextual formalism, to facilitate type checking in the presence of yet unknown receiver classes. That is co-contextual requirements need to be merged and removed to deduce the well-typedness of a given program and conditions are an additional structure to maintain the relations between the required classes and the declared ones. We have constructed the co-contextual typing rules of FJ and described the translation from the original typing rules using the dualism technique. We have proven the correctness of the co-contextual type checker, by proving the typing equivalence between FJ and co-contextual FJ. We have described all theorems required to prove this equivalence. Finally, we have shown how co-contextual type checking facilitates incremental type checking. We implemented an incremental co-contextual type checker, which gave high speedups compared to javac on 1243 FJ classes and on real java programs,

up to 500 SLOCs.

To evaluate the expressiveness of our dualism technique, we have introduced co-contextual FGJ, an extension of the previous type system with generics. Generics are a powerful language construct used in many OO languages, where methods and classes are parameterized over types. Applying the dualism was not trivial in the presence of generics, because it introduces new constraints and changes the operations of merge and remove. For example, the removal of a field requirement in the presence of generics, required knowledge of the position of the field's type variable in the declaration of the generic parameters of the receiver class.

Additional complexity was added, since we considered generics with bounded types, i.e., generic type variables that must adhere to the class hierarchy and are not applicable to be used with all types. The contextual formulation of FGJ introduces well-formedness rules and a new context for type bounds Δ , in addition to the constructs of FJ. To feature these additional constructs, we systematically applied dualism. That is, we have built co-contextual rules for well-formedness and a dual construct to the context Δ . We have introduced the bounded requirements BR , together with operations that are dual to the operations on Δ .

We constructed the co-contextual typing rules for FGJ. In this translation, the most difficult rules turned out to be the typing rules for method invocation and method declaration. For example, the rule for method invocation introduced many unification variables for the generic types and their bounds, which had to be related to each other via constraints. Hence, the translation was very involved in terms of size of the premises and elements of the judgements that need to be considered. However, the actual translation from the contextual type systems was systematic and eased due to our dualism technique.

In Chapter 11, we further extended co-contextual FJ with method overloading, which was as challenging as generics, yet for different reasons. Method overloading allows a method with the same name to have different signatures. The parameter types used in a method invocation then determine which implementation is used. A method signatures can vary in the arity of the method, or different parameter or return types. We adapted the structural layout of the method requirement to allow a selection of the overloaded method from a set of types, i.e., a multi-type. The introduced set of types has been termed *mltype* and represents a mapping from classes to sets of method signatures. The *mltype* is populated when encountering declarations corresponding to a required method. To choose the most specific signature of a method, we used a minimal selection, which operates on multi-types.

We have described the operations on method requirements in the presence of method overloading. The merge operation was adapted to update the set of conditions (*cond*) using type equalities and inequalities on the parameter types. This adaptation was necessary to indicate that methods are not only matched via their receiver class and name, but also their (overloaded) types. In addition, the arity of method signatures needed to be considered, since requirements can be merged only if their arguments have the same arity and types. The remove operation required the most intricate change, because the of the minimal selection on multi-types. In the co-contextual type systems a removal signifies a declaration has been found that matches the requirements of the method invocation.

However, overloading required the most specific type to be deduced from all overloaded methods. Hence, all method signatures that correspond to the required method and are declared in the class hierarchy of the receiver class need to be provided.

Finally, we have discussed the implications of the multi-types and minimal selections for the efficiency of the co-contextual type checker. Given that the requirements removal has to wait on all methods in a class hierarchy, we cannot discharge requirements gradually. We have discussed the changes required to the co-contextual type checker to deal with method overloading efficiently.

Part IV.

Conclusions and Future directions

14. Conclusions and Future directions

This chapter presents the conclusions drawn from our work and a summary of our contributions (Sec. 14.1) and discusses possible future directions for the work (Sec. 14.2).

14.1. Conclusions

This thesis has proposed a novel method for systematically constructing co-contextual type systems from traditional ones. The technique we used to realize the translation from contextual to co-contextual type systems is dualism. A co-contextual type system removes all contexts and replaces them by the dual structures of *requirements*. This enables the co-contextual type checker to avoid the coordination and remove the dependencies between subexpressions of a given program, which makes them well-suited for incremental, parallel, and compositional type checking. In the following we summarize the contributions presented in this thesis.

Co-Contextual Type Checkers for PCF and Extensions of PCF

First, we have constructed co-contextual type systems for functional languages, where we considered PCF as a representative language. We replaced the typing context by the dual structure of *context requirements*, which are propagated bottom-up. Operations on the requirements are dual to the operations on the typing context. We have built a co-contextual type checker for PCF. To evaluate our technique, we extended co-contextual PCF with records, parametric polymorphism and structural subtyping. We had to add types and constraints to support the new language features, e.g., in the case of records and parametric polymorphism. Moreover, co-contextual parametric polymorphism introduced universally quantified unification variables to support unknown universal types. Besides, we had to change the operations on the context requirements to support structural subtyping.

A complex extension to PCF was let-polymorphism because variables can have different bindings in different parts of the program. In order to support the different binding, we introduced partial type schemas, which are dual to the type schemas found in the traditional type system. The partial type schemas are generated and instantiated using generalization and instantiation constraints. To correlate the program variables of the let binding to its instances in the let body, we introduced compatibility constraints. To evaluate the co-contextual let-polymorphic type checker, we implemented the Haskell list library.

In conclusion, the initial idea of removing contexts proved to be a very challenging endeavor. The introduction of the dualism technique was pivotal in the systematic translation to a co-contextual type system. Once a dual structure has been found, e.g.,

14. Conclusions and Future directions

requirements as dual to the context, the translation is revolving around the operations on that structure. Also operations have duals, e.g., removing a requirement as dual to adding type bindings to the context, yet their semantics can be very challenging.

Having an initial understanding of the operations in the setting of a core language was of great value and we have shown that the technique is applicable for language extensions. Depending on the language construct of the extension changes to the core co-contextual type system can be minimal or challenging. The former was for example true for records. The latter can also be differentiated depending on the features that need to be adapted in the initial phase of constructing the co-contextual type system. For example, structural subtyping required to change the operations, while for parametric polymorphism it was more challenging to find the dual structure for the universal quantification over types. The most challenging feature to co-contextualize was let-polymorphism because the semantics of the let expressions themselves gives rise to complex relations between the let binding and the body. This is reflected in the contextual and naturally in the co-contextual type systems. Hence, finding dual structures proved to be very challenging.

Functional languages lend themselves naturally into the co-contextual setting because in essence their program structure fits into the bottom-up type checking. Even let-polymorphism can be translated into a co-contextual type system without breaking our formalism.

Co-Contextual Type Checkers for FJ and Extensions of FJ

We focused on OO languages to evaluate how co-contextual type systems are constructed in a setting that is less obvious to fit for bottom-up type checking as the functional languages. We used our technique to construct a co-contextual type checker for FJ – as a representative language – by transforming the original typing rules into a form that replaces top-down propagated contexts and class tables with bottom-up propagated *context and class table requirements*. We used dualism to derive the co-contextual FJ from the traditional FJ. However, the translation was more involved in the case of FJ as compared to PCF because we supported the key features of OO languages: nominal typing, subtype polymorphism, and implementation inheritance. Moreover, FJ uses class tables in addition to the typing contexts, which have a more complex structure than the typing contexts. To make the correspondence between the class tables and their requirements, we presented class tables that are gradually extended with information from the class declarations. We described the operations on the class table requirements, which are dual to the operations on the class table. Moreover, to cover the key features of OO languages, we introduced conditional requirements and constraints. In addition, we introduced optional method requirements to feature method overriding.

To evaluate the expressiveness of our technique, we extended FJ with generics and method overloading. These two powerful features were difficult to co-contextualize because we had to introduce new types, constraints, judgements, and sets of requirements to support them. In addition, we changed the operations of merging and removing class table requirements. In the case of generics, we added support for generic types, constructed co-contextual rules for well-formed types and introduced bounded requirements as dual structure to bounded context Δ . To support method overriding in FGJ, which allows

methods to have covariant return types, we introduced optional constraints. Moreover, we had to change the structure of the existing method requirements to support minimal selection in method overloading.

In conclusion, the idea of a contextless type system for an OO language was much more challenging to realize than for the functional language. The rigorous formalism introduced for the latter, was pivotal in this endeavor. It gave an initial setting for the challenge of removing the class table used in the OO language type systems, by introducing a dual class table requirements set. In this respect, the functional context requirement structure was not applicable one-to-one. Instead the class table requirements were extended with conditions to feature nominal typing. Likewise the structure of the constraints needed to be extended with conditions, and all operations required adaptation.

As for the functional languages, the core OO language can be extended and the adaptation of the co-contextual type system has varying challenges for each extension. Overall the extensions in OO are much more involved due to the complexity of their structures and impacts on the type system, e.g., generics add a completely new way of typing.

In contrast to the functional setting, OO languages inherently need global (class table) information to ensure the well-typedness of a given program. In the presence of inheritance and nominal typing, the co-contextual type checker may have to wait with the satisfaction of field or method requirements, if respective declarations are not directly in the receiver class. That is, in the worst case the satisfaction has to wait until all superclass declarations are available. This worst case becomes the standard case in the presence of method overloading, due to the minimal selection. In this respect the construction of the co-contextual type system makes the *necessary* relations between language constructs evident. This allows us to better reason over the theoretical boundaries and the practical optimizations needed to implement type checkers that deal with them efficiently.

Proof of Equivalence Between Contextual and Co-Contextual Type Checkers for PCF, FJ

We have discussed the equivalence between the traditional and co-contextual type systems for both functional and OO languages. We have proven the typing equivalence for expressions in the case of PCF. Furthermore, we have proven the typing equivalence for methods, classes, and programs between FJ and co-contextual FJ. The complete set of proofs, lemmas and theorems is shown in Appendix B.2.

The proofs ensure the correctness of the co-contextual type systems constructed as part of this thesis. That is, we have shown that type checking a program gives the same result in both contextual and co-contextual settings.

Incremental Type Checkers for PCF and FJ, and Performance Evaluations.

In this thesis, we have focused on incremental type checking and described a method for developing efficient incremental type checkers that use the co-contextual type systems. We have implemented co-contextual type checkers for PCF and FJ, and extensions of them.

To obtain an optimal implementation for functional languages we primarily used incremental and continuous solving of constraints. In addition, we have applied memoization

14. Conclusions and Future directions

to avoid recomputing derivations for unchanged subexpressions. Obtaining an optimal implementation was more challenging for OO languages. The optimizations performed for functional languages were still applicable, but needed to be adapted. In addition, we have introduced further optimizations. For example, we have added in-depth merging of class table requirements to reduce the overall number of requirements.

For PCF the performance evaluation has shown that the co-contextual type checker has a performance comparable to the standard contextual type checker and incrementalization improved the performance significantly.

For FJ we have focused on measuring the worst case scenarios to understand the viability of the co-contextual type checkers in OO languages. To this end, we have compared the performance of our incremental co-contextual FJ type checkers not only to the performance of the original FJ, but also to `javac`. Our performance evaluation was realized on synthesized programs, with up to 1243 FJ classes and on real java programs, with up to 500 SLOCs. We have shown that even in the worst case the incremental co-contextual FJ is 6-times faster compared to `javac`.

In conclusion, our performance evaluations have confirmed the notion that functional languages lend themselves to co-contextual type checking and the incrementalization yielded better results. However, OO languages can be incrementalized efficiently by applying optimizations. We have discussed several general purpose optimizations in the course of this thesis, e.g., continuous constraint solving. In addition, we have shown how optimizations that are specific to OO languages can yield further improvements.

14.2. Future Directions

An interesting direction would be to investigate co-contextual formulations of type systems with more powerful programming language features. For example, Scala introduces traits, which require a linearization step, or existential types, which allow to abstract over the existence of a type as part of the program. In this respect also Haskell's use of type classes is of great interest. Formulating these type systems co-contextually will yield new insights into the necessary relations that are evident in these language constructs. Thus allowing us to better reason over the theoretical necessities and the practicalities of dealing with them efficiently in incremental type checkers. Furthermore, we expect that a larger selection of language features will provide evidence of recurring patterns when co-contextualizing them. We have conducted preliminary experiments for type classes, based on the formalization by Wadler et al. [WB89] and a follow up work by Odersky et al. [OWW95]. In our preliminary results, we have introduced conditions to the context requirements, i.e., similar to the conditions in class table requirements, to co-contextualize type classes.

As an immediate step, the extension of the co-contextual type systems to support the full syntax of a functional or OO language is of great value for evaluations in a practical setting. Supporting a full programming language will ease the evaluation, for example of an incremental type checker, by using real programs that are abundantly available. In this respect, we can consider Haskell for functional languages and Java for

OO languages. We have already started to work on Haskell and made progress towards a full co-contextual Haskell. We have investigated the Haskell syntax¹ and have formulated co-contextual typing rules of a subset of Haskell expressions², e.g., list comprehension, case, do, alternative, arithmetic sequences. We have a complete parser of the entire Haskell syntax in our project, but translating all type rules is a significant effort. Nevertheless, this development will lead to a fine-grained incremental type checker for Haskell and will allow more insights into the scalability of co-contextual type checkers and further practical optimizations.

Another step towards incorporating the co-contextual type checker into real world compilers is to improve the performance by exploring additional optimizations. Especially during our experiments with OO languages, we have identified further areas of optimization. For example, to optimize the layout for class declarations (e.g., following the inheritance hierarchy or the package structure) and to reshuffle the layout during incremental type checking in order to keep frequently changing classes as close to the root as possible. Furthermore, we have identified areas for method overloading that can be beneficial to optimize. For example, to gradually refine minimal selections from a single class to the whole class hierarchy, or to mark hierarchies as complete when the root class is determined. As discussed previously, a larger set of languages and features may yield patterns for constructing co-contextual type systems. The same may be expected for optimizations such that recurring patterns in co-contextual type systems may require accompanying recurring optimizations.

Furthermore, it would be interesting to explore co-contextual type checking with parallelization. Parallelization, like incrementalization, can provide significant performance boosts, but is not trivial to apply. Besides avoiding coordination between different subexpressions and removing dependencies between them, parallelization also requires efficient strategies for distributing the syntax tree to and collecting the (sub-)results from multiple workers while keeping the coordination overhead minimal. We have started to investigate strategies for clustering and assessed which would be best applicable to the syntax trees of different programs given our co-contextual setting.

Finally, we deem co-contextual translations of type systems that go beyond the traditional functional and OO languages of very high interest. In this respect concurrent and distributed systems have seen a number of advanced type system proposals to analyze various behavioral properties. Distributed systems are heterogeneous and open, i.e., consisting of smaller parts (processes, components) that may be constructed by different parties and overall configurations may change to incorporate new parts or replace existing ones. Thus, type systems seem a desirable method to provide strong guarantees. The guaranteed behavioral properties range from input/output modes, over race conditions and deadlocks, to information flows and are typically expressed in (variants of) the π -calculus (see [IK01] for a summary).

To deal with the complexity of distributed systems in type systems it seems natural to aim for a compositional technique. Co-contextual type systems provide an inherently

¹<https://www.haskell.org/onlinereport/haskell2010/haskellch10.html>

²<https://ghc.haskell.org/trac/ghc/browser/ghc/compiler/typecheck/TcExpr.hs>

14. *Conclusions and Future directions*

compositional type system that yields a set of requirements and constraints without requiring a global view upfront. Thus, different parts of a distributed system can be typed in isolation and results from different parts may be combined to ensure the overall well-typedness. Furthermore, recent works in distributed systems also explore dynamic reconfigurations, i.e., changes to the overall configuration at runtime, to accommodate for the openness of distributed systems. It is especially in this setting that an incremental co-contextual type checker can yield large benefits, i.e., to allow the deduction of strong guarantees in a running system without impacting the overall performance of the system itself. The π -calculus underlying the type systems for distributed systems is different from the PCF and FJ considered in this thesis. Hence, a co-contextual translation comes with new challenges.

Bibliography

- [ABB⁺09] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *Transactions on Programming Languages and Systems (TOPLAS)*, 32(1), 2009.
- [ABH02] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 247–259, New York, NY, USA, 2002. ACM.
- [ACG92] Isabelle Attali, Jacques Chazarain, and Serge Gilette. Incremental evaluation of natural semantics specifications. *Logical Foundations of Computer Science*, pages 87–104, 1992.
- [ACV96] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 396–409, New York, NY, USA, 1996. ACM.
- [ADDZ05] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for java-like languages. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 26–37. ACM, 2005.
- [ALZ02a] D. Ancona, G. Lagorio, and E. Zucca. True separate compilation of java classes. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '02, pages 189–200, New York, NY, USA, 2002. ACM.
- [ALZ02b] Davide Ancona, Giovanni Lagorio, and Elena Zucca. A formal framework for java separate compilation. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 609–636, Berlin, Heidelberg, 2002. Springer-Verlag.
- [AN91] Shail Aditya and Rishiyur S. Nikhil. Incremental polymorphism. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 379–405. ACM, 1991.

Bibliography

- [ARO14] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 233–249, New York, NY, USA, 2014. ACM.
- [AZ04] Davide Ancona and Elena Zucca. Principal typings for java-like languages. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 306–317, New York, NY, USA, 2004. ACM.
- [Bac81] John Backus. History of programming languages. chapter The History of Fortran I, II, and III, pages 25–74. ACM, New York, NY, USA, 1981.
- [BBG⁺63] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language algol 60. *Commun. ACM*, 6(1):1–17, January 1963.
- [BCV09] Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Featherweight java with dynamic and static overloading. *Sci. Comput. Program.*, 74(5-6):261–278, March 2009.
- [BS86] Rolf Bahlke and Gregor Snelting. The PSG system: From formal language definitions to interactive programming environments. *Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):547–576, 1986.
- [Car88a] L. Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 70–79, New York, NY, USA, 1988. ACM.
- [Car88b] L. Cardelli. Typechecking dependent types and subtypes. In *Lecture Notes in Computer Science on Foundations of Logic and Functional Programming*, pages 45–57, New York, NY, USA, 1988. Springer-Verlag New York, Inc.
- [Car02] Magnus Carlsson. Monads for incremental computing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 26–35, New York, NY, USA, 2002. ACM.
- [CF58] Haskell B. Curry and Robert Feys. Combinatory logic, volume 1. North-Holland Publishing Company, 1958.
- [Chi01] Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 193–204. ACM, 2001.
- [Chr13] David Raymond Christiansen. Bidirectional typing rules: A tutorial, 2013.

- [Chu40] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5(2):56–68, 06 1940.
- [CLR98] Luc A. Chamberland, Sharon F. Lymer, and Arthur G. Ryman. IBM VisualAge for Java. *IBM Systems Journal*, 37(3):386–408, 1998.
- [Des84] Thierry Despeyroux. Executable specification of static semantics. In *Semantics of Data Types*, volume 173 of *LNCS*. Springer, 1984.
- [DK13] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of International Conference on Functional Programming (ICFP)*, 2013.
- [DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.
- [DRT81] Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 105–116. ACM, 1981.
- [EBK⁺15] Sebastian Erdeweg, Oliver Bračevac, Edlira Kuci, Mathias Krebs, and Mira Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015. to appear.
- [FT90] John Field and Tim Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 307–322, New York, NY, USA, 1990. ACM.
- [Gir71] Jean-Yves Girard. Une extension de l'interprétation de gödel a l'analyse, et son application a l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63 – 92. Elsevier, 1971.
- [GMW79] Michael J. C Gordon, 1934 Milner, Robin, and Christopher P Wadsworth. *Edinburgh LCF : a mechanised logic of computation*. Berlin ; New York : Springer-Verlag, 1979. Includes index.
- [HPHF14] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 156–166, New York, NY, USA, 2014. ACM.

Bibliography

- [IK01] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the π -calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 128–141, New York, NY, USA, 2001. ACM.
- [Jim96] Trevor Jim. What are principal typings and what are they good for? In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 42–53. ACM, 1996.
- [JW86] Gregory F. Johnson and Janet A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 44–57. ACM, 1986.
- [Kah87] Gilles Kahn. Natural semantics. In *Proceedings of Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 22–39. Springer, 1987.
- [LT95] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, February 1995.
- [Mee83] Lambert G. L. T. Meertens. Incremental polymorphic type checking in B. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 265–275. ACM, 1983.
- [MEK⁺14] Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. i3QL: Language-integrated live data views. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 417–432. ACM, 2014.
- [Mil] Robin Milner.
- [MS10] Weiyu Miao and Jeremy G. Siek. Incremental type-checking for type-reflective metaprograms. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 167–176. ACM, 2010.
- [OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 135–146, New York, NY, USA, 1995. ACM.
- [Pie02] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.
- [Pot01] François Pottier. Simplifying subtyping constraints: A theory. *Information and Computation*, 170(2):153–183, 2001.
- [PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

- [SA93] Zhong Shao and Andrew W. Appel. Smartest recompilation. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 1993.
- [SB07] Ajeet Shankar and Rastislav Bodik. Ditto: Automatic incrementalization of data structure invariant checks (in java). volume 42, pages 310–319, 06 2007.
- [SEV16] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. Inca: A dsl for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 320–331, New York, NY, USA, 2016. ACM.
- [SH86] Gregor Snelting and Wolfgang Henhagl. Unification in many-sorted algebras as a device for incremental semantic analysis. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 229–235. ACM, 1986.
- [Sne91] Gregor Snelting. The calculus of context relations. *Acta Informatica*, 28(5):411–445, 1991.
- [TFK⁺11] Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.*, 33(3):9:1–9:47, May 2011.
- [Tic86] Walter F. Tichy. Smart recompilation. *Transactions on Programming Languages and Systems (TOPLAS)*, 1986.
- [TKB03] Frank Tip, Adam Kiezun, and Dirk Bäumler. Refactoring for generalization using type constraints. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '03*, pages 13–26, New York, NY, USA, 2003. ACM.
- [Uni05] Cornell University. Let-polymorphism. In *Lecture Notes, CS, Cornell*, 2005. <http://www.cs.cornell.edu/courses/cs312/2005sp/lectures/rec22.asp>.
- [WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89*, pages 60–76, New York, NY, USA, 1989. ACM.
- [Wel02] J. B. Wells. The essence of principal typings. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.
- [Wir71] Niklaus Wirth. The programming language pascal. *Acta Inf.*, 1:35–63, 1971.
- [WKV⁺13] Guido Wachsmuth, Gabriël D. P. Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. A language independent task engine for incremental name and type analysis. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 8225 of *LNCS*, pages 260–280, 2013.

Appendix

A. Equivalence of Contextual and Co-Contextual PCF

In this appendix we provide the proof to Theorem 1, showing that our formulations of contextual and co-contextual PCF are equivalent.

Recall that we call a syntactic entity ground if it does not contain unification variables and we write $\Gamma \supseteq R$ if $\Gamma(x) = R(x)$ for all $x \in \text{dom}(R)$.

Lemma 1. *Let $\text{merge}(R_1, R_2) = R|_C$, $\Gamma \supseteq \sigma_1(R_1)$, $\Gamma \supseteq \sigma_2(R_2)$, and $\sigma_1(R_1)$ and $\sigma_2(R_2)$ be ground. Then $\sigma_1 \circ \sigma_2$ solves C .*

Proof. By the definition of merge , $C = \{R_1(x) = R_2(x) \mid x \in \text{dom}(R_1) \cap \text{dom}(R_2)\}$. Since $\Gamma \supseteq \sigma_i(R_i)$, we know $\Gamma(x) = \sigma_i(R_i(x))$ for all $x \in \text{dom}(R_i)$. In particular, $\Gamma(x) = \sigma_1(R_1(x)) = \sigma_2(R_2(x))$ for all $x \in \text{dom}(R_1) \cap \text{dom}(R_2)$. Thus, $\sigma_1 \circ \sigma_2$ solves C because $(\sigma_1 \circ \sigma_2)(R_1(x)) = \sigma_1(R_1(x)) = \sigma_2(R_2(x)) = (\sigma_1 \circ \sigma_2)(R_2(x))$ for all $x \in \text{dom}(R_1) \cap \text{dom}(R_2)$, because $\sigma_1(R_1)$ and $\sigma_2(R_2)$ are ground. \square

Theorem 1 (Equivalence of contextual PCF and co-contextual PCF). *A program e is typeable in contextual PCF if and only if it is typeable in co-contextual PCF:*

$\Gamma \vdash e : T \mid C$ and $\text{solve}(C) = \sigma$ such that
 $\sigma(T)$ and $\sigma(\Gamma)$ are ground

if and only if

$e : T' \mid C' \mid R$ and $\text{solve}(C') = \sigma'$ such that
 $\sigma'(T')$ and $\sigma'(R)$ are ground

If e is typeable in contextual and co-contextual PCF as above, then $\sigma(T) = \sigma'(T')$ and $\sigma(\Gamma) \supseteq \sigma'(R)$.

Proof. We first show that typeability in contextual PCF entails typeability in co-contextual PCF (\Rightarrow) with matching types and contexts/requirements. We proceed by structural induction on e .

- Case n with $\Gamma \vdash n : T \mid C$.

By inversion, $T = \text{Num}$ and $C = \emptyset$.

We choose $T' = \text{Num}$, $C' = \emptyset$, $R = \emptyset$, and $\sigma' = \emptyset$.

Then $e : T' \mid C' \mid R$ holds, $\sigma(T) = \text{Num} = \sigma'(T')$, $\Gamma \supseteq \emptyset = \sigma'(R)$, and $\sigma'(T')$ and $\sigma'(R)$ are ground.

- Case x with $\Gamma \vdash x : T \mid C$.

By inversion, $\Gamma(x) = T$ and $C = \emptyset$.

A. Equivalence of Contextual and Co-Contextual PCF

We choose $T' = U$, $C' = \emptyset$, $R = \{x : U\}$, and $\sigma' = \{U \mapsto T\}$.

Then $e : T' \mid C' \mid R$ holds, $\sigma(T) = T = \sigma'(U) = \sigma'(T')$, $\Gamma \supseteq \{x : T\} = \sigma'(R)$, $\sigma'(T')$ and $\sigma'(R)$ are ground because T is ground.

- Case $\lambda x : T_1. e$ with $\Gamma \vdash \lambda x : T_1. e : T \mid C$.

By inversion, $x : T_1; \Gamma \vdash e : T_2 \mid C_e$, $T = T_1 \rightarrow T_2$, and $C = C_e$.

Let $\text{solve}(C) = \sigma$.

By IH, $e : T_2' \mid C_e' \mid R_e$ with $\text{solve}(C_e') = \sigma_e'$, $\sigma_e(T_2) = \sigma_e'(T_2')$, $(x : T_1; \Gamma) \supseteq \sigma_e'(R_e)$, $\sigma_e'(T_2')$ is ground, and $\sigma_e'(R_e)$ is ground.

We choose $T' = T_1 \rightarrow T_2'$ and $R = R_e - x$.

- If $x \in \text{dom}(R_e)$, then $R_e(x) = U_e$ for some U_e . We choose $C' = C_e' \cup \{T_1 = R_e(x)\}$ and $\sigma' = \sigma_e' \circ \{U_e \mapsto T_1\}$.

Then $e : T' \mid C' \mid R$ holds and σ' solves C_e' and $\sigma'(T_1) = T_1 = \sigma'(U_e) = \sigma'(R_e(x))$. Moreover, $\sigma(T) = T_1 \rightarrow \sigma(T_2) = T_1 \rightarrow \sigma'(T_2') = \sigma'(T')$, $\Gamma \supset \sigma'(R_e - x)$, $\sigma'(T')$ is ground because T_1 and $\sigma'(T_2')$ are ground, and $\sigma'(R)$ is ground because $\sigma_e'(R_e)$ is ground.

- If $x \notin \text{dom}(R_e)$, we choose $C' = C_e'$ and $\sigma' = \sigma_e'$.

Then $e : T' \mid C' \mid R$ holds, σ' solves C_e' , $\sigma(T) = T_1 \rightarrow \sigma(T_2) = T_1 \rightarrow \sigma'(T_2') = \sigma'(T')$, $\Gamma \supset \sigma'(R_e - x)$, $\sigma'(T')$ is ground because T_1 and $\sigma'(T_2')$ are ground, and $\sigma'(R)$ is ground because $\sigma_e'(R_e)$ is ground.

- Case $e_1 e_2$ with $\Gamma \vdash e_1 e_2 : T \mid C$.

By inversion, $\Gamma \vdash e_1 : T_1 \mid C_1$, $\Gamma \vdash e_2 : T_2 \mid C_2$, $T = U$, and $C = C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow U\}$.

Let $\text{solve}(C) = \sigma$, which also solves C_1 , C_2 , and $T_1 = T_2 \rightarrow U$ such that $\sigma(U)$ is ground.

By IH for $i \in \{1, 2\}$, $e_i : T_i' \mid C_i' \mid R_i$ with $\text{solve}(C_i') = \sigma_i'$, $\sigma(T_i) = \sigma'(T_i')$, $\Gamma \supseteq \sigma_i'(R_i)$, $\sigma_i'(T_i')$ is ground, and $\sigma_i'(R_i)$ is ground.

Let $\text{merge}(R_1, R_2) = R|_{C_r'}$. We choose $T' = U$, $C' = C_1' \cup C_2' \cup \{T_1' = T_2' \rightarrow U\} \cup C_r'$, and $\sigma' = \sigma_1' \circ \sigma_2' \circ \{U \mapsto \sigma(U)\}$.

Then $e_1 e_2 : T' \mid C' \mid R$ holds and σ' solves C' because it solves C_1' , C_2' , C_r' (by Lemma 2), and $\sigma'(T_1') = \sigma(T_1) = \sigma(T_2) \rightarrow \sigma(U) = \sigma'(T_2') \rightarrow \sigma'(U) = \sigma'(T_2' \rightarrow U)$.

Moreover, $\sigma'(T') = \sigma'(U) = \sigma(U) = \sigma(T)$, $\Gamma \supseteq \sigma'(R_1) \cup \sigma'(R_2) \supseteq \sigma'(R)$ by the definition of merge , $\sigma'(T')$ is ground because $\sigma(U)$ is ground, and $\sigma'(R)$ is ground because $\sigma_i'(R_i)$ are ground.

- Case $e_1 + e_2$ with $\Gamma \vdash e_1 + e_2 : T \mid C$.

By inversion, $\Gamma \vdash e_1 : T_1 \mid C_1$, $\Gamma \vdash e_2 : T_2 \mid C_2$, $T = \text{Num}$, and $C = C_1 \cup C_2 \cup \{T_1 = \text{Num}, T_2 = \text{Num}\}$.

Let $\text{solve}(C) = \sigma$, which also solves $C_1, C_2, T_1 = \text{Num}$, and $T_2 = \text{Num}$.

By IH for $i \in \{1, 2\}$, $e_i : T'_i \mid C'_i \mid R_i$ with $\text{solve}(C'_i) = \sigma'_i$, $\sigma'_i(T'_i) = \sigma'(T'_i)$, $\Gamma \supseteq \sigma'(R_i)$, $\sigma'(T'_i)$ is ground, and $\sigma'(R_i)$ is ground.

Let $\text{merge}(R_1, R_2) = R|_{C'_r}$. We choose $T' = \text{Num}$, $C' = C'_1 \cup C'_2 \cup \{T'_1 = \text{Num}, T'_2 = \text{Num}\} \cup C'_r$, and $\sigma' = \sigma'_1 \circ \sigma'_2$.

Then $e_1 + e_2 : T' \mid C' \mid R$ and σ' solves C' because it solves C'_1, C'_2, C'_r (by Lemma 2), and $\sigma'(T'_i) = \sigma(T_i) = \text{Num}$.

Moreover, $\sigma'(T') = \text{Num} = \sigma(T)$, $\Gamma \supseteq \sigma'(R_1) \cup \sigma'(R_2) \supseteq \sigma'(R)$ by the definition of merge , $\sigma'(T') = \text{Num}$ is ground, and $\sigma'(R)$ is ground because $\sigma'_i(R_i)$ are ground.

- Case $\text{if0 } e_1 \ e_2 \ e_3$ with $\Gamma \vdash \text{if0 } e_1 \ e_2 \ e_3 : T \mid C$.

By inversion for $i \in \{1, 2, 3\}$, $\Gamma \vdash e_i : T_i \mid C_i$, $T = T_2$, and $C = C_1 \cup C_2 \cup C_3 \cup \{T_1 = \text{Num}, T_2 = T_3\}$.

Let $\text{solve}(C) = \sigma$, which also solves $C_1, C_2, C_3, T_1 = \text{Num}$, and $T_2 = T_3$.

By IH, $e_i : T'_i \mid C'_i \mid R_i$ with $\text{solve}(C'_i) = \sigma'_i$, $\sigma'_i(T'_i) = \sigma(T_i)$, $\Gamma \supseteq \sigma'_i(R_i)$, $\sigma'(T'_i)$ is ground, and $\sigma'(R_i)$ is ground.

Let $\text{merge}(R_2, R_3) = R_{2,3}|_{C'_{2,3}}$, $\text{merge}(R_1, R_{2,3}) = R_{1,2,3}|_{C'_{1,2,3}}$. We choose $R = R_{1,2,3}$, $T' = T'_2$, $C' = C'_1 \cup C'_2 \cup C'_3 \cup \{T'_1 = \text{Num}, T'_2 = T'_3\} \cup C'_{2,3} \cup C'_{1,2,3}$, and $\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3$.

Then $\text{if0 } e_1 \ e_2 \ e_3 : T' \mid C' \mid R$ and σ' solves C' because it solves $C'_1, C'_2, C'_3, C'_{2,3}$ (by Lemma 2), $C'_{1,2,3}$ (by Lemma 2), and $\sigma'(T'_1) = \sigma(T_1) = \text{Num}$ as well as $\sigma'(T'_2) = \sigma(T_2) = \sigma(T_3) = \sigma'(T'_3)$.

Moreover, $\sigma'(T') = \sigma'(T'_2) = \sigma(T_2) = \sigma(T)$, $\Gamma \supseteq \sigma'(R_1) \cup \sigma'(R_2) \cup \sigma'(R_3) \supseteq \sigma'(R_1) \cup \sigma'(R_{2,3}) \supseteq \sigma'(R_{1,2,3})$ by the definition of merge , $\sigma'(T')$ is ground because $\sigma'(T'_2)$ is ground, and $\sigma'(R)$ is ground because $\sigma'_i(R_i)$ are ground.

- Case $\text{fix } e$ with $\Gamma \vdash \text{fix } e : T \mid C$.

By inversion, $\Gamma \vdash e : T_e \mid C_e$, $T = U$, and $C = C_e \cup \{T_e = U \rightarrow U\}$.

Let $\text{solve}(C) = \sigma$, which solves C_e and $T_e = U \rightarrow U$ such that $\sigma(U)$ is ground.

By IH, $e : T'_e \mid C'_e \mid R_e$ with $\text{solve}(C'_e) = \sigma'_e$, $\sigma'_e(T'_e) = \sigma(T_e)$, $\Gamma_e \supseteq \sigma'(R_e)$, $\sigma'(T'_e)$ is ground, and $\sigma'(R_e)$ is ground.

We choose $R = R_e$, $T' = U$, $C' = C'_e \cup \{T'_e = U \rightarrow U\}$, and $\sigma' = \sigma'_e \circ \{U \mapsto \sigma(U)\}$.

Then $\text{fix } e : T' \mid C' \mid R$ and σ' solves C' because it solves C'_e and $\sigma'(T'_e) = \sigma(T_e) = \sigma(U \rightarrow U) = \sigma(U) \rightarrow \sigma(U) = \sigma'(U) \rightarrow \sigma'(U) = \sigma'(U \rightarrow U)$.

Moreover, $\sigma'(T') = \sigma'(U) = \sigma(U) = \sigma(T)$, $\Gamma = \Gamma_e \supseteq \sigma'(R_e) = \sigma'(R)$, $\sigma'(T')$ is ground because $\sigma(U)$ is ground, and $\sigma'(R)$ is ground because $\sigma'(R_e)$ is ground.

Next we show that typeability in co-contextual PCF entails typeability in contextual PCF (\Leftarrow) with matching types and contexts/requirements. We again proceed by structural induction on e .

A. Equivalence of Contextual and Co-Contextual PCF

- Case n with $n : T' \mid C' \mid R$.
 By inversion, $T' = \text{Num}$, $C' = \emptyset$, and $R = \emptyset$.
 Let $\text{solve}(C') = \sigma'$. We choose $\Gamma = \emptyset$, $T = \text{Num}$, $C = \emptyset$, $\sigma = \emptyset$.
 Then $\Gamma \vdash e : T \mid C$ holds, $\sigma(T) = \text{Num} = \sigma'(T')$, $\Gamma = \emptyset = \sigma'(R)$, and $\sigma(T)$ is ground.
- Case x with $x : T' \mid C' \mid R$.
 By inversion, $T' = U$, $C' = \emptyset$, and $R = \{x : U\}$.
 Let $\sigma'(U) = T_x$ for some T_x that we know is ground.
 We choose $\Gamma = x : T_x; \emptyset$, $T = T_x$, $C = \emptyset$, and $\sigma = \emptyset$.
 Then $\Gamma \vdash e : T \mid C$ holds, $\sigma(T) = T_x = \sigma'(U)$, $\Gamma = (x : T_x; \emptyset) \supseteq \{x : T_x\} = \sigma'(R)$, and $\sigma(T)$ is ground because T_x is ground.
- Case $\lambda x : T_1. e$ with $\lambda x : T_1. e : T' \mid C' \mid R$.
 By inversion, $e : T_2' \mid C_e' \mid R_e$, $T' = T_1 \rightarrow T_2'$, and $R = R_e - x$ for some T_2' , C_e' , and R_e .
 Let $\text{solve}(C') = \sigma'$, which also solves C_e' .
 By IH, $\Gamma_e \vdash T_2 : e \mid C_e$ with $\text{solve}(C_e) = \sigma_e$, $\sigma_e(T_2) = \sigma'(T_2')$, $\Gamma_e \supseteq \sigma'(R_e)$, and $\sigma_e(T_2)$ is ground. The latter entails $\Gamma_e(x) = \sigma'(R_e(x))$ for all $x \in \text{dom}(R_e)$.
 We choose $T = T_1 \rightarrow T_2$, $C = C_e$, and $\sigma = \sigma_e$ such that $\sigma(T) = T_1 \rightarrow \sigma(T_2) = T_1 \rightarrow \sigma'(T_2') = \sigma'(T')$ and $\sigma(T)$ is ground because T_1 and $\sigma_e(T_2)$ are ground.
 - If $x \in \text{dom}(R_e)$, then $(T_1 = R_e(x)) \in C'$ and $T_1 = \sigma'(R_e(x)) = \Gamma_e(x)$. Thus, by swapping $\Gamma_e = (x : T_1; \Gamma)$ for some Γ such that $\Gamma \vdash e : T \mid C$ holds, σ solves C , and $\Gamma \supseteq \Gamma_e - x \supseteq \sigma'(R_e) - x = \sigma'(R_e - x)$.
 - If $x \notin \text{dom}(R_e)$, the x is not free in e and we get $x : T_1; (\Gamma_e - x) \vdash T_2 : e \mid C_e$ by strengthening and weakening. We choose $\Gamma = \Gamma_e - x$ such that (i) holds and $\Gamma = \Gamma_e - x \supseteq \sigma'(R_e) - x = \sigma'(R_e - x)$.
- Case $e_1 e_2$ with $e_1 e_2 : T' \mid C' \mid R$.
 By inversion, $e_1 : T_1' \mid C_1' \mid R_1$, $e_2 : T_2' \mid C_2' \mid R_2$, $\text{merge}(R_1, R_2) = R|_{C_r'}$, $T' = U$, and $C' = C_1' \cup C_2' \cup \{T_1' = T_2' \rightarrow U\} \cup C_r'$.
 Let $\text{solve}(C') = \sigma'$, which also solves C_1' , C_2' , C_r' , and $T_1' = T_2' \rightarrow U$ such that $\sigma'(U)$ is ground.
 By IH for $i \in \{1, 2\}$, $\Gamma_i \vdash e_i : T_i \mid C_i$ with $\text{solve}(C_i) = \sigma_i$, $\sigma_i(T_i) = \sigma'(T_i')$, $\Gamma_i \supseteq \sigma'(R_i)$, and $\sigma_i(T_i)$ is ground.
 We choose $\Gamma = \Gamma_1; \Gamma_2$, $T = U$, $C = C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow U\}$, and $\sigma = \sigma_1 \circ \sigma_2 \circ \{U \mapsto \sigma'(U)\}$.
 Since $\sigma_i(\Gamma_i) \supseteq \sigma'(R_i)$, Γ only extends Γ_1 and Γ_2 with variables that are not free in e_1 and e_2 , respectively. Thus, $\Gamma \vdash e_i : T_i \mid C_i$ and $\Gamma \vdash e_1 e_2 : U \mid C$. σ solves C

because it solves C_1 and C_2 and $\sigma(T_1) = \sigma'(T'_1) = \sigma'(T'_2 \rightarrow U) = \sigma'(T'_2) \rightarrow \sigma'(U) = \sigma(T_2) \rightarrow \sigma(U) = \sigma(T_2 \rightarrow U)$.

Moreover, $\sigma(T) = \sigma(U) = \sigma'(U) = \sigma'(T')$, $\Gamma = \Gamma_1; \Gamma_2 \supseteq \sigma'(R_1) \cup \sigma'(R_2) \supseteq \sigma'(R)$ by the definition of *merge*, and $\sigma(T)$ is ground because $\sigma'(U)$ is ground.

- Case $e_1 + e_2$ with $e_1 + e_2 : T' \mid C' \mid R$.

By inversion for $i \in \{1, 2\}$, $e_i : T'_i \mid C'_i \mid R_i$, $\text{merge}(R_1, R_2) = R|_{C'_r}$, $T' = \text{Num}$, and $C' = C'_1 \cup C'_2 \cup \{T'_1 = \text{Num}, T'_2 = \text{Num}\} \cup C'_r$.

Let $\text{solve}(C') = \sigma'$, which also solves $C'_1, C'_2, C'_r, T'_1 = \text{Num}$, and $T'_2 = \text{Num}$.

By IH, $\Gamma_i \vdash e_i : T_i \mid C_i$ with $\text{solve}(C_i) = \sigma_i$, $\sigma_i(T_i) = \sigma'(T'_i)$, $\Gamma_i \supseteq \sigma'(R_i)$, and $\sigma_i(T_i)$ is ground.

We choose $\Gamma = \Gamma_1; \Gamma_2$, $T = \text{Num}$, $C = C_1 \cup C_2 \cup \{T_1 = \text{Num}, T_2 = \text{Num}\}$, and $\sigma = \sigma_1 \circ \sigma_2$.

Since $\sigma_i(\Gamma_i) \supseteq \sigma'(R_i)$, Γ only extends Γ_1 and Γ_2 with variables that are not free in e_1 and e_2 , respectively. Thus, $\Gamma \vdash e_i : T_i \mid C_i$ and $\Gamma \vdash e_1 + e_2 : \text{Num} \mid C$.

σ solves C because it solves C_1 and C_2 and $\sigma(T_i) = \sigma'(T'_i) = \text{Num}$.

Moreover, $\sigma(T) = \text{Num} = \sigma'(T')$, $\Gamma = \Gamma_1; \Gamma_2 \supseteq \sigma'(R_1) \cup \sigma'(R_2) \supseteq \sigma'(R)$ by the definition of *merge*, and $\sigma(T) = \text{Num}$ is ground.

- Case $\text{if0 } e_1 \ e_2 \ e_3$ with $\text{if0 } e_1 \ e_2 \ e_3 : T' \mid C' \mid R$.

By inversion for $i \in \{1, 2, 3\}$, $e_i : T'_i \mid C'_i \mid R_i$, $\text{merge}(R_2, R_3) = R_{2,3}|_{C_{2,3}}$, $\text{merge}(R_1, R_{2,3}) = R_{1,2,3}|_{C_{1,2,3}}$, $T' = T$, $R = R_{1,2,3}$, and $C' = C'_1 \cup C'_2 \cup C'_3 \cup \{T'_1 = \text{Num}, T'_2 = T'_3\} \cup C_{2,3} \cup C_{1,2,3}$.

Let $\text{solve}(C') = \sigma'$, which also solves $C'_1, C'_2, C'_3, C_{2,3}, C_{1,2,3}, T'_1 = \text{Num}$, and $T'_2 = T'_3$.

By IH, $\Gamma_i \vdash e_i : T_i \mid C_i$ with $\text{solve}(C_i) = \sigma_i$, $\sigma_i(T_i) = \sigma'(T'_i)$, $\Gamma_i \supseteq \sigma'(R_i)$, and $\sigma_i(T_i)$ is ground.

We choose $\Gamma = \Gamma_1; \Gamma_2; \Gamma_3$, $T = T_2$, $C = C_1 \cup C_2 \cup C_3 \cup \{T_1 = \text{Num}, T_2 = T_3\}$, and $\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3$.

Since $\sigma_i(\Gamma_i) \supseteq \sigma'(R_i)$, Γ only extends Γ_1, Γ_2 , and Γ_3 with variables that are not free in e_1, e_2 , and e_3 , respectively. Thus, $\Gamma \vdash e_i : T_i \mid C_i$ and $\Gamma \vdash \text{if0 } e_1 \ e_2 \ e_3 : T_2 \mid C$.

σ solves C because it solves C_1, C_2 , and C_3 and $\sigma(T_1) = \sigma'(T'_1) = \text{Num}$ as well as $\sigma(T_2) = \sigma'(T'_2) = \sigma'(T'_3) = \sigma(T_3)$.

Moreover, $\sigma(T) = \sigma_2(T_2) = \sigma'(T'_2) = \sigma'(T')$, $\Gamma = \Gamma_1; \Gamma_2; \Gamma_3 \supseteq \sigma'(R_1) \cup \sigma'(R_2) \cup \sigma'(R_3) \supseteq \sigma'(R_1) \cup \sigma'(R_{2,3}) \supseteq \sigma'(R_{1,2,3})$ by the definition of *merge*, and $\sigma(T)$ is ground because $\sigma_2(T_2)$ is ground.

- Case $\text{fix } e$ with $\text{fix } e : T' \mid C' \mid R$.

By inversion, $e : T'_e \mid C'_e \mid R_e$, $T' = U$, $C' = C'_e \cup \{T'_e = U \rightarrow U\}$, and $R = R_e$.

A. Equivalence of Contextual and Co-Contextual PCF

Let $\text{solve}(C') = \sigma'$, which solves C'_e and $T'_e = U \rightarrow U$ such that $\sigma'(U)$ is ground.

By IH, $\Gamma_e \vdash e : T_e \mid C_e$ with $\text{solve}(C_e) = \sigma_e$, $\sigma_e(T_e) = \sigma'(T'_e)$, $\Gamma_e \supseteq \sigma'(R_e)$, and $\sigma(T_e)$ is ground.

We choose $\Gamma = \Gamma_e$, $T = U$, $C = C_e \cup \{T_e = U \rightarrow U\}$, and $\sigma = \sigma_e \circ \{U \mapsto \sigma'(U)\}$.

Then $\Gamma \vdash \text{fix } e : T \mid C$ and σ solves C because it solves C_e and $\sigma(T_e) = \sigma'(T'_e) = \sigma'(U \rightarrow U) = \sigma'(U) \rightarrow \sigma'(U) = \sigma(U) \rightarrow \sigma(U) = \sigma(U \rightarrow U)$. Moreover, $\sigma(T) = \sigma(U) = \sigma'(U) = \sigma'(T')$, $\Gamma = \Gamma_e \supseteq \sigma'(R_e) = \sigma'(R)$, and $\sigma(T)$ is ground because $\sigma'(U)$ is ground.

□

B. Equivalence of Contextual and Co-Contextual FJ

B.1. Auxiliary definitions; merge, add, remove

We give the definition of $merge_{CR}$ for all cases of the clause definition ¹.

$$\begin{aligned}
merge_{CR}(CR_1, CR_2) &= CR|_S \\
\text{where } CR &= \{((CR_1 \setminus (\overline{T_1}.extends : T_2, cond_1 \cup \overline{T_1}.init(\overline{T_1}), cond_1 \cup \overline{T_1}.f : T_2, cond_1 \\
&\quad \cup \overline{T_1}.m : \overline{T_1} \rightarrow T_2, cond_1) \cup ((CR_2 \setminus (\overline{T_2}.extends : T_3, cond_2 \\
&\quad \cup \overline{T_2}.init(\overline{T_2}), cond_2 \cup \overline{T_2}.f : T_3, cond_2 \cup \overline{T_2}.m : \overline{T_2} \rightarrow T_3, cond_2) \\
&\quad \cup CR_e \cup CR_k \cup CR_f \cup CR_m)\} \\
S &= S_e \cup S_k \cup S_f \cup S_m \\
\text{where } CR_e &= \{(T_1.extends : T'_1, cond_1 \cup (T_1 \neq T_2)), (T_2.extends : T'_2, cond_2 \cup \\
&\quad (T_1 \neq T_2)), (T_1.extends : T'_1, (cond_1 \cup cond_2 \cup (T_1 = T_2)) \\
&\quad | (T_1.extends : T'_1, cond_1) \in CR_1 \wedge \\
&\quad (T_2.extends : T'_2, cond_2) \in CR_2\} \\
S_e &= \{(T'_1 = T'_2 \text{ if } T_1 = T_2) | (T_1.extends : T'_1, cond_1) \in CR_1 \wedge \\
&\quad (T_2.extends : T'_2, cond_2) \in CR_2\} \\
\text{where } CR_k &= \{(T_1.init(\overline{T_1}), cond_1 \cup (T_1 \neq T_2)) \cup (T_2.init(\overline{T_2}), cond_2 \cup (T_1 \neq T_2)) \\
&\quad (T_1.init(\overline{T_1}), cond_1 \cup cond_2 \cup (T_1 = T_2)) | (T_1.init(\overline{T_1}), cond_1) \in CR_1 \\
&\quad \wedge (T_2.init(\overline{T_2}), cond_2) \in CR_2\} \\
S_k &= \{(\overline{T_1} = \overline{T_2} \text{ if } T_1 = T_2) | (T_1.init(\overline{T_1}), cond_1) \in CR_1 \wedge \\
&\quad (T_2.init(\overline{T_2}), cond_2) \in CR_2\}
\end{aligned}$$

¹Merge operation for optional methods is the same as merge for methods.

B. Equivalence of Contextual and Co-Contextual FJ

$$\begin{aligned}
\text{where } CR_f &= \{(T_1.f : T'_1, \text{cond}_1 \cup (T_1 \neq T_2)) \cup (T_2.f : T'_2, \text{cond}_2 \cup (T_1 \neq T_2)) \\
&\quad \cup (T_1.f : T'_1, \text{cond}_1 \cup \text{cond}_2 \cup (T_1 = T_2)) \mid (T_1.f : T'_1, \text{cond}_1) \in CR_1 \\
&\quad \wedge (T_2.f : T'_2, \text{cond}_2) \in CR_2\} \\
S_f &= \{(T'_1 = T'_2 \text{ if } T_1 = T_2) \mid (T_1.f : T'_1, \text{cond}_1) \in CR_1 \wedge \\
&\quad (T_2.f : T'_2, \text{cond}_2) \in CR_2\} \\
\text{where } CR_m &= \{(T_1.m : \overline{T}_1 \rightarrow T'_1, \text{cond}_1 \cup (T_1 \neq T_2)) \cup (T_2.m : \overline{T}_2 \rightarrow T'_2, \text{cond}_2 \cup \\
&\quad (T_1 \neq T_2)) \cup (T_1.m : \overline{T}_1 \rightarrow T'_1, \text{cond}_1 \cup \text{cond}_2 \cup (T_1 = T_2)) \mid (T_1.m : \overline{T}_1 \rightarrow T'_1, \text{cond}_1) \in CR_1 \wedge (T_2.m : \overline{T}_2 \rightarrow T'_2, \text{cond}_2) \in CR_2\} \\
S_m &= \{(T'_1 = T'_2 \text{ if } T_1 = T_2) \cup (\overline{T}_1 = \overline{T}_2 \text{ if } T_1 = T_2) \mid (T_1.m : \overline{T}_1 \rightarrow T'_1, \text{cond}_1) \in CR_1 \wedge (T_2.m : \overline{T}_2 \rightarrow T'_2, \text{cond}_2) \in CR_2\}
\end{aligned}$$

Next, we define the add and remove operations for all cases of the clause definition.

$$\begin{aligned}
\text{addExt}(\text{class } C \text{ extends } D, CT) &= (C \text{ extends } D) \cup CT \\
\text{removeExt}(\text{class } C \text{ extends } D, CR) &= CR'|_S \\
\text{where } CR' &= \{(T.\text{extends} : T', \text{cond} \cup (T \neq C)) \mid (T.\text{extends} : T', \text{cond}) \in CR\} \\
&\quad \cup \{(T.m : \overline{T} \rightarrow T', \text{cond} \cup (T \neq C)) \\
&\quad \cup (D.m : \overline{T} \rightarrow T', \text{cond} \cup (T = C)) \mid (T.m : \overline{T} \rightarrow T', \text{cond}) \in CR\} \\
&\quad \cup \{(T.m : \overline{T} \rightarrow T', \text{cond} \cup (T \neq C))_{opt} \\
&\quad \cup (D.m : \overline{T} \rightarrow T', \text{cond} \cup (T = C))_{opt} \\
&\quad \mid (T.m : \overline{T} \rightarrow T', \text{cond})_{opt} \in CR\} \\
&\quad \cup \{(T.f : T', \text{cond} \cup (T \neq C)) \cup (D.f : T', \text{cond} \cup (T = C)) \\
&\quad \mid (T.f : T', \text{cond}) \in CR\} \\
S &= \{(T' = D \text{ if } T = C) \mid (T.\text{extends} : T', \text{cond}) \in CR\} \\
\text{addCtor}(C, (\overline{D} \ \overline{g}, \overline{C} \ \overline{f}), CT) &= (C.\text{init}(\overline{D}; \overline{C})) \cup CT \\
\text{removeCtor}(C, (\overline{D} \ \overline{g}, \overline{C} \ \overline{f}), CR) &= CR'|_S \\
\text{where } CR' &= \{(T.\text{init}(\overline{T})), \text{cond} \cup (T \neq C)) \mid (T.\text{init}(\overline{T})), \text{cond}) \in CR\} \\
&\quad \cup (CR \setminus \overline{(T.\text{init}(\overline{T})), \text{cond}}) \\
S &= \{(\overline{T} = \overline{D} \ \overline{C} \text{ if } T = C) \mid (T.\text{init}(\overline{T})), \text{cond}) \in CR\}
\end{aligned}$$

$$\text{addFs}(C, \overline{C_f} \overline{f}, CT) = \overline{C.f : C_f} \cup CT$$

$$\text{removeF}(C, C_f f, CR) = CR'|_S$$

$$\text{where } CR' = \{(T.f : T', \text{cond} \cup (T \neq C)) \mid (T.f : T', \text{cond}) \in CR\}$$

$$\cup (CR \setminus \overline{(T.f : T', \text{cond})})$$

$$S = \{(T' = C_f \text{ if } T = C) \mid (T.f : T', \text{cond}) \in CR\}$$

$$\text{removeFs}(C, \overline{C_f} \overline{f}, CR) = CR'|_S$$

$$\text{where } CR' = \{CR_f \mid (C_f f) \in \overline{C_f} \overline{f} \wedge \text{removeF}(CR, C, C_f f) = CR_f|_{S_f}\}$$

$$S = \{S_f \mid (C_f f) \in \overline{C_f} \overline{f} \wedge \text{removeF}(CR, C, C_f f) = CR_f|_{S_f}\}$$

$$\text{addMs}(C, \overline{M}, CT) = \overline{C.m : \overline{C} \rightarrow C'} \cup CT$$

$$\text{removeM}(C, C' m(\overline{C} \overline{x}) \{\text{return } e\}, CR) = CR'|_S$$

$$\text{where } CR' = \{(T.m : \overline{T} \rightarrow T', \text{cond} \cup (T \neq C)) \mid (T.m : \overline{T} \rightarrow T', \text{cond}) \in CR\}$$

$$\cup (CR \setminus (T.m : \overline{T} \rightarrow T', \text{cond}))$$

$$S = \{(T' = C' \text{ if } T = C) \cup (\overline{T} = \overline{C} \text{ if } T = C) \mid (T.m : \overline{T} \rightarrow T', \text{cond}) \in CR\}$$

$$\text{removeMs}(C, \overline{M}, CR) = CR'|_S$$

$$\text{where } CR' = \{CR_m \mid (C' m(\overline{C} \overline{x}) \{\text{return } e\}) \in \overline{M} \wedge$$

$$\text{removeM}(C, C' m(\overline{C} \overline{e}) \{\text{return } e\}, CR) = CR_m|_{S_m}\}$$

$$S = \{S_m \mid (C' m(\overline{C} \overline{x}) \{\text{return } e\}) \in \overline{M} \wedge$$

$$\text{removeM}(C, C' m(\overline{C} \overline{x}) \{\text{return } e\}, CR) = CR_m|_{S_m}\}$$

$$\text{removeOptM}(C, C' m(\overline{C} \overline{x}) \{\text{return } e\}, CR) = CR'|_S$$

$$\text{where } CR' = \{(T.m : \overline{T} \rightarrow T', \text{cond} \cup (T \neq C))_{\text{opt}} \mid (T.m : \overline{T} \rightarrow T', \text{cond})_{\text{opt}} \in CR\}$$

$$\cup (CR \setminus (T.m : \overline{T} \rightarrow T', \text{cond}_{\text{opt}}))$$

$$S = \{(T' = C' \text{ if } T = C) \cup (\overline{T} = \overline{C} \text{ if } T = C) \mid (T.m : \overline{T} \rightarrow T', \text{cond})_{\text{opt}} \in CR\}$$

$$\text{removeOptMs}(C, \overline{M}, CR) = (CR' \cup (CR \setminus CR'))|_S$$

$$\text{where } CR' = \{CR_m \mid (C' m(\overline{C} \overline{x}) \{\text{return } e\}) \in \overline{M} \wedge$$

$$\text{removeOptM}(CR, C, C' m(\overline{C} \overline{e}) \{\text{return } e\}) = CR_m|_{S_m}\}$$

$$S = \{S_m \mid (C' m(\overline{C} \overline{x}) \{\text{return } e\}) \in \overline{M} \wedge$$

$$\text{removeOptM}(CR, C, C' m(\overline{C} \overline{x}) \{\text{return } e\}) = CR_m|_{S_m}\}$$

B.2. Equivalence of Contextual and Co-Contextual FJ

We describe a detailed proof of typing equivalence between FJ and co-contextual FJ. Co-contextual FJ is constraint based type system. We present the formal definitions for substitution, and Figures B.1, B.2 give formal definition how to retrieve the immediate subclass relation Σ from rep. class table, and a list of class declaration. That is, a projection from class table/list of declarations to a set of tuples, which represent the relation between two classes in an extends clause.

$$\begin{array}{c}
 \frac{}{\text{projExt}(\emptyset) = \emptyset} \quad \frac{}{\text{projExt}(C \text{ extends } D) = (C, D)} \\
 \\
 \frac{}{\text{projExt}(C.f : C') = \emptyset} \quad \frac{}{\text{projExt}(C.\text{init}(\overline{C})) = \emptyset} \\
 \\
 \frac{}{\text{projExt}(C.m() : \overline{C} \rightarrow C') = \emptyset} \\
 \\
 \frac{\text{projExt}(CTcls_1) = \Sigma_1 \quad \text{projExt}(CTcls_2) = \Sigma_2}{\text{projExt}(CTcls_1 \cup CTcls_2) = \Sigma_1 \cup \Sigma_2}
 \end{array}$$

Figure B.1.: Projection of Class Table to Extends.

$$\begin{array}{c}
 \frac{}{\text{projExt}(\emptyset) = \emptyset} \quad \frac{}{\text{projExt}(\text{class } C \text{ extends } D \{ \overline{C} \ \overline{f}; \ K \ \overline{M} \}) = (C, D)} \\
 \\
 \frac{\text{projExt}(L_1) = \Sigma_1 \quad \text{projExt}(L_2) = \Sigma_2}{\text{projExt}(L_1; L_2) = \Sigma_1 \cup \Sigma_2}
 \end{array}$$

Figure B.2.: Projection of Class Declarations to Extends.

Definition 1 (Subtyping relative to Σ). *Let Σ be a binary relation on class names, C, D class names. Then C is a subtype of D relative to Σ ($C <_{\Sigma} D$), if and only if $(C, D) \in \Sigma^*$, where Σ^* is the reflexive, transitive closure of Σ .*

Definition 2 (Substitution σ). *Given sets of context and class requirements R, CR , σ is a set of mappings from class variables to class types, i.e., $\sigma = \{U \mapsto C \mid U \in \text{fresh}U(R) \cup \text{fresh}U(CR)\}$.*

Definition 3 (Constraint Satisfaction). *Let s be a constraint on class types, σ a substitution from class variables to class types, Σ a binary relation on class names. The pair (Σ, σ) satisfies s ($\text{sat}(\Sigma, \sigma, s)$) if and only if one of the following holds:*

1. If $s = (T <: T')$, then $T\sigma <:_\Sigma T'\sigma$.
2. If $s = (T = T')$, then $T\sigma = T'\sigma$.
3. If $s = (T = T' \text{ if } cond)$ and for all $s' \in cond$, $\text{sat}(\Sigma, \sigma, s')$ then $T\sigma = T'\sigma$.
4. If $s = (T \neq T')$, then $T\sigma \neq T'\sigma$.
5. If $s = (T \not<:_\Sigma T')$, then $T\sigma \not<:_\Sigma T'\sigma$.

Assumption 1 (Properties of solve). Let Σ be a binary relation on class names, S a set of constraints on class types:

1. $\text{solve}(\Sigma, S)$ terminates.
2. If $\text{solve}(\Sigma, S) = \sigma$. Then for all $s \in S$, $\text{sat}(\Sigma, \sigma, s)$.
3. If $\text{solve}(\Sigma, S) = \perp$. Then there exists $s \in S$, where $\text{sat}(\Sigma, \sigma, s)$ does not hold.

Definition 4 (Ground context requirement). $\sigma(R)$ is ground, if for all $(x : T) \in R$ then $\sigma(T)$ is ground.

Definition 5 (Ground class table requirements). $\sigma(CR)$ is ground, if for all $(CReq, cond) \in CR$ then $\sigma(CReq)$ is ground and $\sigma(cond)$ is ground.

Definition 6 (Ground class requirement).

$$\sigma(CReq) \text{ ground} = \begin{cases} \sigma(T.\text{extends} : T') \text{ ground} & \text{if } (CReq) = (T.\text{extends}T') \wedge \\ & \sigma(T), \sigma(T') \text{ ground} \\ \sigma(T.f : T') \text{ ground} & \text{if } (CReq) = (T.f : T') \wedge \\ & \sigma(T), \sigma(T') \text{ ground} \\ \sigma(T.m : \bar{T} \rightarrow T') \text{ ground} & \text{if } (CReq) = (T.m : \bar{T} \rightarrow T') \wedge \quad (\text{B.1}) \\ & \sigma(T), \sigma(\bar{T}) \text{ ground} \\ & \wedge \sigma(T') \text{ ground} \\ \sigma(T.\text{init} : \bar{T}) \text{ ground} & \text{if } (CReq) = (T.\text{init}(\bar{T})) \wedge \\ & \sigma(T), \sigma(\bar{T}) \text{ ground} \end{cases}$$

Definition 7 (Ground conditions). $\sigma(cond)$ is ground, if for all $(T = T'), (T'' \neq T^*) \in cond$ then $\sigma(T), \sigma(T'), \sigma(T''), \sigma(T^*)$ are ground.

Definition 8 (Ground Solution σ). For a given type T , a set of constraints S , where $\sigma = \text{solve}(S)$, we lift substitution σ to sets of context requirements R , class requirements CR and σ is a ground solution if:

- 1) $\sigma(T)$ is ground
- 2) $\sigma(R)$ is ground

B. Equivalence of Contextual and Co-Contextual FJ

$$\begin{array}{c}
\text{FIELD-LOOKUP} \frac{C.f_i : C_i \in \text{fields}(C, CT)}{\text{field}(f_i, C, CT) = C_i} \quad \text{EXTENDS} \frac{(C.\text{extends} = D) \in CT}{\text{extends}(C, CT) = D} \\
\\
\text{S-EXTEND} \frac{(C.\text{extends} = D) \in CT}{CT \text{ satisfies } (C.\text{extends} : D, \text{cond})} \\
\\
\text{S-CONSTRUCTOR} \frac{\text{fields}(C, CT) = \overline{C.f : C_f}}{CT \text{ satisfies } (C.\text{init}(\overline{C_f}), \text{cond})} \\
\\
\text{S-FIELD} \frac{\text{field}(f, C, CT) = C'}{CT \text{ satisfies } (C.f : C', \text{cond})} \\
\\
\text{S-METHOD} \frac{\text{if } \text{mtype}(m, C, CT) = \overline{C} \rightarrow C'}{CT \text{ satisfies } (C.m : \overline{C} \rightarrow C', \text{cond})} \\
\\
\text{SATISFIES} \frac{(\text{cond hold} \Rightarrow CT \text{ satisfy } (CReq, \text{cond})) \quad \forall (CReq, \text{cond}) \in CR}{CT \text{ satisfies } CR}
\end{array}$$

Figure B.3.: Judgment for Satisfies.

3) $\sigma(CR)$ is ground

The two first rules of Figure.B.3 define the field lookup and extends lookup. The other rules formally define the relation between the class table and class table requirements. We assume that class table requirements are ground.

Lemma 2. *Let $\text{merge}_R(R_1, R_2) = R|_S$, $\Gamma \supseteq \sigma_1(R_1)$, $\Gamma \supseteq \sigma_2(R_2)$, and $\sigma_1(R_1)$, $\sigma_2(R_2)$ are ground. Then $\sigma_1 \circ \sigma_2$ solves S .*

Proof. By the definition of merge_R , $S = \{R_1(x) = R_2(x) \mid x \in \text{dom}(R_1) \cap \text{dom}(R_2)\}$. Since $\Gamma \supseteq \sigma_i(R_i)$, we know $\Gamma(x) = \sigma_i(R_i(x))$ for all $x \in \text{dom}(R_i)$. In particular, $\Gamma(x) = \sigma_1(R_1(x)) = \sigma_2(R_2(x))$ for all $x \in \text{dom}(R_1) \cap \text{dom}(R_2)$. Thus, $\sigma_1 \circ \sigma_2$ solves C because $(\sigma_1 \circ \sigma_2)(R_1(x)) = \sigma_1(R_1(x)) = \sigma_2(R_2(x)) = (\sigma_1 \circ \sigma_2)(R_2(x))$ for all $x \in \text{dom}(R_1) \cap \text{dom}(R_2)$, because $\sigma_1(R_1)$ and $\sigma_2(R_2)$ are ground. \square

Lemma 3. *Let $\text{merge}_{CR}(CR_1, CR_2) = CR|_S$, $\sigma_1(CR_1)$, $\sigma_2(CR_2)$ are ground, and CT satisfies $\sigma_1(CR_1)$, CT satisfies $\sigma_2(CR_2)$. Then $\sigma_1 \circ \sigma_2$ solves S .*

Proof. By the definition of merge_{CR} , $S = S_c \cup S_e \cup S_k \cup S_f \cup S_m$, where $S_f = \{(T'_1 = T'_2 \text{ if } T_1 = T_2) \mid (T_1.f : T'_1, \text{cond}_1) \in CR_1 \wedge (T_2.f : T'_2, \text{cond}_2) \in CR_2\}$.

Since CT satisfies $\sigma_i(CR_i)$, we know $\sigma_i(T_i.f : T'_i, \text{cond}_i) \in CT$, for all $f \in \text{dom}(CR_i)$, where $\sigma_i(\text{cond}_i)$ hold. In particular, for all $f \in \text{dom}(CR_1) \cap \text{dom}(CR_2)$, where $(T_1.f : T'_1, \text{cond}_1) \in CR_1$, $(T_2.f : T'_2, \text{cond}_2) \in CR_2$, $\sigma_1(T'_1) = \sigma_2(T'_2)$ if $\sigma_1(T_1) = \sigma_2(T_2)$. Thus,

$\sigma_1 \circ \sigma_2$ solves S because $(\sigma_1 \circ \sigma_2)(T'_1) = \sigma_1(T'_1) = \sigma_2(T'_2) = (\sigma_1 \circ \sigma_2)(T'_2)$, if $\sigma_1(T_1) = \sigma_2(T_2)$, because $\sigma_1(CR_1)$ and $\sigma_2(CR_2)$ are ground.

The same procedure we follow for methods, i.e., a given method m that we find a match in $CR_1(C)$, and $CR_2(C)$, S_m is the set of constraints for the method as result of unifying return type and types of the parameters from the two different class requirements(CR_1, CR_2). \square

Lemma 4. *If CT satisfies $\sigma_1(CR_1)$, $\sigma_1(CR_1)$ is ground, and CT satisfies $\sigma_2(CR_2)$, $\sigma_2(CR_2)$ is ground, then CT satisfies $\sigma(CR)$, where $\sigma = \sigma_1 \circ \sigma_2$ and $CR_S = merge_{CR}(CR_1, CR_2)$.*

Proof. First we have to show that the new set of constraints S generated from merging is solvable, and this holds by Lemma 3.

Then we show that CT satisfies $\sigma(CR)$. For sake of brevity we consider clauses common in both requirements sets CR_1 and CR_2 . Let us consider the field f , such that $(T_1.f : T'_1, cond_1) \in CR_1$ and $(T_2.f : T'_2, cond_2)$, and by assumption we have that CT satisfies $\sigma_1((T_1.f : T'_1, cond_1))$ and CT satisfies $\sigma_2(T_2.f : T'_2, cond_2)$. By the definition of $merge_{CR}$ the conditions of these two requirements are updated, i.e., $(T_1.f : T'_1, cond_1 \cup (T_1 \neq T_2))$ and $(T_2.f : T'_2, cond_2 \cup (T_1 \neq T_2))$, and a new requirement is added, i.e., $(T_1.f : T'_1, cond_1 \wedge cond_2 \cup (T_1 = T_2))$. Suppose that CT satisfies the three of the new and updated requirements, namely all their conditions should hold by rule SATISFIES, but this is contradiction, because two types cannot be at the same time not equal and equal. Therefore there are two possibilities:

- 1) either the conditions of the updated field requirements hold, i.e., $(T_1.f : T'_1, cond_1 \cup (T_1 \neq T_2))$, $(T_2.f : T'_2, cond_2 \cup (T_1 \neq T_2))$, and $(T_1 \neq T_2)$ holds.
 - 2) or the conditions of the new field requirement hold, i.e., $(T_1.f : T'_1, cond_1 \wedge cond_2 \cup (T_1 = T_2))$, and $(T_1 = T_2)$ holds.
- If 1) is possible then CT satisfies $\sigma_1 \circ \sigma_2(T_1.f : T'_1, cond_1 \cup (T_1 \neq T_2) \cup T_2.f : T'_2, cond_2 \cup (T_1 \neq T_2))$ because by assumption CT satisfies $\sigma_1((T_1.f : T'_1, cond_1))$ and CT satisfies $\sigma_2(T_2.f : T'_2, cond_2)$. The new class requirement $(T_1.f : T'_1, cond_1 \wedge cond_2 \cup (T_1 = T_2))$ is satisfiable by default since one of its conditions $(T_1 = T_2)$ does not hold, namely is not a valid requirement.
 - If 2) is possible then CT satisfies $\sigma_1 \circ \sigma_2(T_1.f : T'_1, cond_1 \wedge cond_2 \cup (T_1 = T_2))$, because $(T_1 = T_2)$, CT satisfies $\sigma_1((T_1.f : T'_1, cond_1))$ and CT satisfies $\sigma_2(T_2.f : T'_2, cond_2)$. The updated class requirements $(T_1.f : T'_1, cond_1 \cup (T_1 \neq T_2))$ and $(T_2.f : T'_2, cond_2 \cup (T_1 \neq T_2))$ are satisfiable by default since one of their conditions $(T_1 \neq T_2)$ does not hold, namely are not valid requirements.

As a result CT satisfies the resulting set of requirements after merging for the given field f .

The same we argue for methods, optional methods, current class, and extend clauses. \square

B. Equivalence of Contextual and Co-Contextual FJ

Proposition 1 (Independent derivation in co-contextual type checking). *Given a set of otherwise independent derivations of class requirement $CR = \{CR_1 \cup \dots \cup CR_n\}$, $\forall i, j \in [1..n]. \text{fresh}U(CR_i) \cap \text{fresh}U(CR_j) = \emptyset$, where $\text{fresh}U(CR_i) = \{U_1^i, \dots, U_n^i\}$*

Proof. It is straightforward by the rules and how the type checking is performed, i.e., for every rules of the type checking we always introduce fresh class names U , therefore U s in one derivation do not appear to another independent derivation. \square

Corollary 1 (Associative feature for substitution). *Given CR, σ_1 and σ_2 then it holds that $(\sigma_1 \circ \sigma_2)(CR) = (\sigma_2 \circ \sigma_1)(CR)$*

Proof. Follows directly from Proposition 1. \square

Definition 9 (Correspondence relation for expressions). *Given judgments $\Gamma; CT \vdash e : C$, $e : T \mid S \mid R \mid CR$, and $\text{solve}(\Sigma, S) = \sigma$, where $\text{projExt}(CT) = \Sigma$. The correspondence relation between Γ and R , CT and CR , written $(C, \Gamma, CT) \triangleright \sigma(T, R, CR)$, is defined as:*

- a) $C = \sigma(T)$
- b) $\Gamma \supseteq \sigma(R)$
- c) CT satisfies $\sigma(CR)$

Theorem 10 (Equivalence of expressions: \Rightarrow). *Given e, C, Γ, CT , if $\Gamma; CT \vdash e : C$, then there exists $T, S, R, CR, \Sigma, \sigma$, where $\text{projExt}(CT) = \Sigma$ and $\text{solve}(\Sigma, S) = \sigma$, such that $e : T \mid S \mid R \mid CR$ holds, σ is a ground solution and $(C, \Gamma, CT) \triangleright \sigma(T, R, CR)$ holds.*

Proof. We proceed by induction on the typing judgment of expression e .

- Case T-VAR with $\Gamma; CT \vdash x : C$.

By inversion, $\Gamma(x) = C$.

Let U fresh, $S = \emptyset$, $R = \{x : U\}$, $CR = \emptyset$ and $\sigma = \{U \mapsto C\}$.

Then $e : C' \mid S' \mid R \mid CR$ holds by rule TC-VAR. Since $S = \emptyset$, then σ solves S . σ is ground solution because:

- 1) $\sigma(U)$ is ground because $\sigma(U) = C$.
- 2) $R = \{x : U\}$ and $\sigma = \{U \mapsto C\}$ implies $\sigma(R) = \{x : C\}$ is ground.
- 3) $CR = \emptyset$ implies that $\sigma(CR) = \emptyset$ is ground.

The correspondence relation holds because;

- a) $C = \sigma(U)$
- b) Since $\Gamma(x) = C$ by inversion, then $\Gamma \supseteq \{x : C\} = \sigma(R)$.
- c) $CR = \emptyset$ and $\sigma(CR) = \emptyset$ implies that CT satisfies $\sigma(CR)$.

B.2. Equivalence of Contextual and Co-Contextual FJ

- Case T-FIELD with $\Gamma; CT \vdash e.f_i : C_i$.

By inversion, $\Gamma; CT \vdash e : C_e$ and $\text{field}(f_i, C_e, CT) = C_i$. By IH, $e : T'_e \mid S_e \mid R_e \mid CR_e$, where $\text{solve}(\text{projExt}(CT), S_e) = \sigma_e$, $\sigma_e(T'_e)$, $\sigma_e(R_e)$, $\sigma_e(CR_e)$ are ground and the correspondence relation holds, i.e., $C_e = \sigma_e(T'_e)$, $\Gamma \supseteq \sigma_e(R_e)$, CT satisfies $\sigma_e(CR_e)$.

Let U be fresh, $CR|_{S_f} = \text{merge}_{CR}(CR_e, (T'_e.f_i : U, \emptyset))$, $S = S_e \cup S_f$ and $\sigma = \{U \mapsto C_i\} \circ \sigma_e$.

Then $e.f_i : U \mid S \mid R_e \mid CR$ holds by rule TC-FIELD .

σ solves S because it solves S_e and S_f as shown below:

- $\text{solve}(\text{projExt}(CT), S_e) = \sigma_e$ by IH and $\sigma = \{U \mapsto C_i\} \circ \sigma_e$ implies σ solves S_e
- $\sigma_e(CR_e)$ is ground by IH.
 - (*) $\sigma(T'_e.f_i : U, \emptyset)$ is ground, because $\sigma(T'_e.f_i : U) = (\sigma(T'_e).f_i : \sigma(U)) = (C_e.f_i : C_i)$ and $C_e.f_i : C_i$ is ground.
 - CT satisfies $\sigma_e(CR_e)$ by IH.
 - (**) CT satisfies $\sigma(T'_e.f : U, \emptyset)$ because $\text{field}(f_i, C_e, CT) = C_i$ hence by rule S-FIELD holds that CT satisfies $(C_e.f : C_i, \emptyset)$, and $\sigma(T'_e.f_i : U) = C_e.f_i : C_i$.
- As a result by Lemma 3 σ solves S_f .

σ is a ground solution because:

- 1) $\sigma(U)$ is ground because $\sigma(U) = C_i$.
- 2) $\sigma(R_e)$ is ground because $\sigma(R_e) = (\{U \rightarrow C_i\} \circ \sigma_e)(R_e) = \{U \rightarrow C_i\}(\sigma_e(R_e))$, since $\sigma_e(R_e)$ is ground by IH then $\{U \rightarrow C_i\}(\sigma_e(R_e)) = \sigma_e(R_e)$, i.e., $\sigma(R_e) = \sigma_e(R_e)$.
- 3) $\sigma(CR_e)$ is ground because $\sigma(CR_e) = (\{U \rightarrow C_i\} \circ \sigma_e)(CR_e) = \{U \rightarrow C_i\}(\sigma_e(CR_e))$, $\{U \rightarrow C_i\}(\sigma_e(CR_e)) = \sigma_e(CR_e)$ because $\sigma_e(CR_e)$ is ground by IH. $\sigma(T'_e.f_i : U, \emptyset)$ is ground by (*). As a result $\sigma(CR)$ is ground by definition of merge_{CR} .

The correspondence relation holds because:

- a) $C_i = \sigma(U)$
- b) $\Gamma \supseteq \sigma(R_e)$, because $\Gamma \supseteq \sigma_e(R_e)$ by IH, and from 2) $\sigma(R_e) = \sigma_e(R_e)$.
- c) CT satisfies $\sigma(CR_e)$, because CT satisfies $\sigma_e(CR_e)$ by IH, and from 3) $\sigma(CR_e) = \sigma_e(CR_e)$. CT satisfies $\sigma(T'_e.f : U, \emptyset)$ by (**). As a result CT satisfies $\sigma(CR_e) \cup \sigma(T'_e.f : U, \emptyset)$, i.e., CT satisfies $\sigma(CR)$ by Lemma 4.

- Case T-INVK with $\Gamma; CT \vdash e.m(\bar{e}) : C$.

By inversion, $\Gamma; CT \vdash e : C_e$, $\text{mtyp}(m, C_e, CT) = \bar{D} \rightarrow C$, $\Gamma; CT \vdash \bar{e} : \bar{C}$ and $\bar{C} <: \bar{D}$.

By IH, $e : T_e \mid S_e \mid R_e \mid CR_e$, where $\text{solve}(\text{projExt}(CT), S_e) = \sigma_e$, $\sigma_e(T_e)$, $\sigma_e(R_e)$, $\sigma_e(CR_e)$ are ground and the correspondence relation hold, i.e, $C_e = \sigma_e(T_e)$, $\Gamma \supseteq \sigma_e(R_e)$, CT satisfies $\sigma_e(CR_e)$.

B. Equivalence of Contextual and Co-Contextual FJ

By IH $\bar{e} : \bar{T} \mid \bar{S} \mid \bar{R} \mid \bar{C}\bar{R}, \forall i \in [1..n]. \text{solve}(\text{projExt}(CT), S_i) = \sigma_i, \sigma_i(T_i), \sigma_i(R_i), \sigma_i(CR_i)$ are ground, and the correspondence relation holds, i.e., $C_i = \sigma_i(T_i), \Gamma \supseteq \sigma_i(R_i), CT$ satisfies $\sigma_i(CR_i)$.

Let U', \bar{U} be fresh, $R|_{S_r} = \text{merge}_R(R_e, R_1, \dots, R_n), CR|_{S_{cr}} = \text{merge}_{CR}(CR_e, CR_1, \dots, CR_n, (T_e.m : \bar{U} \rightarrow U', \emptyset)), S = S_e \cup \bar{S} \cup S_r \cup S_{cr} \cup \{\bar{T} <: \bar{U}\}$ and $\sigma = \{U' \mapsto C\} \circ \{U_i \mapsto D_i\}_{i \in [1..n]} \circ \sigma_e \circ \{\sigma_i\}_{i \in [1..n]}$

Then $e.m(\bar{e}) : U \mid S \mid R \mid CR$ holds by rule TC-INVK.

σ solves S because it solves $S_e, \bar{S}, S_r, S_{cr}$, and $\{\bar{T} <: \bar{U}\}$ as shown below:

- $\text{solve}(\text{projExt}(CT), S_e) = \sigma_e$ and $\sigma = \{U \mapsto C\} \circ \{U_i \mapsto D_i\}_{i \in [1..n]} \circ \sigma_e \circ \{\sigma_i\}_{i \in [1..n]}$ implies that σ solves S_e
- $\{\text{solve}(\text{projExt}(CT), S_i) = \sigma_i\}_{i \in [1..n]}$ and $\sigma = \{U \mapsto C\} \circ \{U_i \mapsto D_i\}_{i \in [1..n]} \circ \sigma_e \circ \{\sigma_i\}_{i \in [1..n]}$ implies that σ solves \bar{S}
- σ solves S_r by Lemma 2.
- $\sigma_e(CR_e), \forall i \in [1..n]. \sigma_i(CR_i)$ are ground by IH.
 (*) $\sigma(T_e.m : \bar{U} \rightarrow U', \emptyset)$ is ground because
 $\sigma(T_e.m : \bar{U} \rightarrow U') = (\sigma(T_e).m : \sigma(\bar{U}) \rightarrow \sigma(U')) = C_e.m : \bar{D} \rightarrow C$ and $C_e.m : \bar{D} \rightarrow C$ is ground.
 CT satisfies $\sigma_e(CR_e), \forall i \in [1..n]. CT$ satisfies $\sigma_i(CR_i)$ by IH.
 (**) CT satisfies $\sigma(T_e.m : \bar{U} \rightarrow U', \emptyset)$ because $\text{mtype}(m, C_e, CT) = \bar{D} \rightarrow C$ hence by rule S-METHOD holds that CT satisfies $(C_e.m : \bar{D} \rightarrow C, \emptyset)$, and $\sigma(T_e.m : \bar{U} \rightarrow U') = C_e.m : \bar{D} \rightarrow C$.
 As a result σ solves S_{cr} by Lemma 3.
- Since $\{\bar{C} <: \bar{D}\}$ holds and $\sigma(\{\bar{T} <: \bar{U}\}) = \{\bar{C} <: \bar{D}\}$, then $\sigma(\{\bar{T} <: \bar{U}\})$ holds

σ is ground solution because

- 1) $\sigma(U')$ is ground because $\sigma(U') = C$
- 2) $\sigma(R_e)$ is ground because $\sigma(R_e) = (\{U' \mapsto C\} \circ \{U_i \mapsto D_i\}_{i \in [1..n]} \circ \sigma_e \circ \{\sigma_i\}_{i \in [1..n]})(R_e) = (\sigma_e \circ \{\sigma_i\}_{i \in [1..n]})(R_e)$ because U', \bar{U} are defined fresh.
 $(\sigma_e \circ \{\sigma_i\}_{i \in [1..n]})(R_e) = (\{\sigma_i\}_{i \in [1..n]} \circ \sigma_e)(R_e)$ by Corollary 1.
 $(\{\sigma_i\}_{i \in [1..n]} \circ \sigma_e)(R_e) = (\{\sigma_i\}_{i \in [1..n]})(\sigma_e(R_e)) = \sigma_e(R_e)$ because $\sigma_e(R_e)$ is ground by IH.
 $\forall i \in [1..n]. \sigma(R_i)$ is ground because $\sigma(R_i) = (\{U' \mapsto C\} \circ \{U_i \mapsto D_i\}_{i \in [1..n]} \circ \sigma_e \circ \{\sigma_i\}_{i \in [1..n]})(R_i) = (\sigma_e \circ \{\sigma_i\}_{i \in [1..n]})(R_i)$ because U', \bar{U} are defined fresh.
 $(\sigma_e \circ \{\sigma_i\}_{i \in [1..n]})(R_i) = (\sigma_e \circ \{\sigma_j\}_{j \in [1..i-1, i+1..n]} \circ \sigma_i)(R_i)$ by Corollary 1.
 $(\sigma_e \circ \{\sigma_j\}_{j \in [1..i-1, i+1..n]})(\sigma_i(R_i)) = \sigma_i(R_i)$ because $\sigma_i(R_i)$ is ground by IH. As a result $\sigma(R)$ is ground by definition of merge_R .
- 3) $\sigma(CR_e)$ is ground because $\sigma(CR_e) = (\{U' \mapsto C\} \circ \{U_i \mapsto D_i\}_{i \in [1..n]} \circ \sigma_e \circ \{\sigma_i\}_{i \in [1..n]})(CR_e) = (\sigma_e \circ \{\sigma_i\}_{i \in [1..n]})(CR_e)$ because U', \bar{U} are defined fresh.
 $(\sigma_e \circ \{\sigma_i\}_{i \in [1..n]})(CR_e) = (\{\sigma_i\}_{i \in [1..n]} \circ \sigma_e)(CR_e)$ by Corollary 1.
 $(\{\sigma_i\}_{i \in [1..n]} \circ \sigma_e)(CR_e) = (\{\sigma_i\}_{i \in [1..n]})(\sigma_e(CR_e)) = \sigma_e(CR_e)$ because $\sigma_e(CR_e)$

is ground by IH .

$\forall i \in [1..n]$. $\sigma(CR_i)$ is ground because $\sigma(CR_i) = (\{U' \mapsto C\} \circ \{U_i \mapsto D_i\}_{i \in [1..n]} \circ \sigma_e \circ \{\sigma_i\}_{i \in [1..n]})(CR_i) = (\sigma_e \circ \{\sigma_i\}_{i \in [1..n]})(CR_i)$ because U' , \bar{U} are defined fresh.

$(\sigma_e \circ \{\sigma_i\}_{i \in [1..n]})(CR_i) = (\sigma_e \circ \{\sigma_j\}_{j \in [1..i-1, i+1..n]} \circ \sigma_i)(CR_i)$ by Corollary 1.

$(\sigma_e \circ \{\sigma_j\}_{j \in [1..i-1, i+1..n]})(\sigma_i(CR_i)) = \sigma_i(CR_i)$ because $\sigma_i(CR_i)$ is ground by IH . $\sigma(T_e.m : \bar{U} \rightarrow U, \emptyset)$ is ground by $(*)$. As a result $\sigma(CR)$ is ground by definition of $merge_{CR}$,

The correspondence relation holds because:

- a) $C = \sigma(U)$
- b) $\Gamma \supseteq \sigma(R_e)$ because $\Gamma \supseteq \sigma_e(R_e)$ by IH , and from 2) $\sigma(R_e) = \sigma_e(R_e)$. $\forall i \in 1 \dots n$. $\Gamma \supseteq \sigma(R_i)$ because $\Gamma \supseteq \sigma_i(R_i)$ by IH , and from 2) $\sigma(R_i) = \sigma_i(R_i)$. As a result $\Gamma \supseteq \sigma(R)$ by definition of $merge_R$.
- c) CT satisfies $\sigma(CR_e)$ because CT satisfies $\sigma_e(CR_e)$, and from 3) $\sigma(CR_e) = \sigma_e(CR_e)$. $\forall i \in 1 \dots n$. CT satisfies $\sigma(CR_i)$ because CT satisfies $\sigma_i(CR_i)$ by IH , and from 3) $\sigma(CR_i) = \sigma_i(CR_i)$. CT satisfies $\sigma(T_e.m : \bar{U} \rightarrow U', \emptyset)$ by $(**)$. As a result CT satisfies $\sigma(CR_e) \cup \sigma(CR_1) \dots \cup \sigma(CR_n) \cup \sigma(T_e.m : \bar{U} \rightarrow U')$, i.e., CT satisfies $\sigma(CR)$ by Lemma 4.

- Case T-NEW with $\Gamma; CT \vdash new\ C(\bar{e}) : C$.

By inversion, $\Gamma; CT \vdash \bar{e} : \bar{C}$, $fields(C, CT) = C.init(\bar{D})$ and $\bar{C} <: \bar{D}$.

By IH , $\bar{e} : \bar{T} \mid \bar{S} \mid \bar{R} \mid \bar{C}\bar{R}$, $\forall i \in 1 \dots n$. $solve(projExt(CT), S_i) = \sigma_i$, $\sigma_i(T_i)$, $\sigma_i(R_i)$, $\sigma_i(CR_i)$ are ground, and the correspondence relation holds, i.e., $C_i = \sigma_i(T_i)$, $\Gamma \supseteq \sigma_i(R_i)$, CT satisfies $\sigma_i(CR_i)$.

Let \bar{U} be fresh, $merge_R(R_1, \dots, R_n) = R|_{S_r}$,

$CR|_{S_{cr}} = merge_{CR}(CR_1, \dots, CR_n, (C.init(\bar{U}, \emptyset)))$. $S = \bar{S} \cup S_r \cup S_{cr} \cup$

$\{\bar{T} <: \bar{U}\}$ and $\sigma = \{U_i \mapsto D_i\}_{i \in [1..n]} \circ \{\sigma_i\}_{i \in [1..n]}$.

Then $C.init(\bar{e}) : C \mid S \mid R \mid CR$ holds by rule TC-NEW.

σ solves S because it solves \bar{S} , S_r , S_{cr} , and $\{\bar{T} <: \bar{U}\}$ as shown below:

- $\{solve(projExt(CT), S_i) = \sigma_i\}_{i \in [1..n]}$ and $\sigma = \{U_i \mapsto D_i\}_{i \in [1..n]} \circ \{\sigma_i\}_{i \in [1..n]}$ implies that σ solves \bar{S}
- σ solves S_r by Lemma 2
- $\forall i \in [1..n]$. $\sigma_i(CR_i)$ are ground by IH .
 $(*)$ $\sigma(C.init(\bar{U}), \emptyset)$ is ground because $\sigma(C.init(\bar{U})) = (\sigma(C).init(\sigma(\bar{U}))) = C.init(\bar{D})$ and $C.init(\bar{D})$ is ground.
 $\forall i \in [1..n]$. CT satisfies $\sigma_i(CR_i)$ by IH .
 $(**)$ CT satisfies $\sigma(C.init(\bar{U}), \emptyset)$ because $fields(C, CT) = \bar{C}.f : \bar{D}$ hence by rule S-CONSTRUCTOR holds that CT satisfies $(C.init(\bar{D}), \emptyset)$, and $\sigma(C.init(\bar{U})) = C.init(\bar{D})$.
 As a result σ solves S_{cr} by Lemma 3.

B. Equivalence of Contextual and Co-Contextual FJ

– Since $\{\overline{C} <: \overline{D}\}$ holds and $\sigma(\{\overline{T} <: \overline{U}\}) = \{\overline{C} <: \overline{D}\}$, then $\sigma(\{\overline{T} <: \overline{U}\})$ holds
 σ is ground solution because:

- 1) $\sigma(C)$ is ground because C is ground.
- 2) $\forall i \in [1..n]. \sigma(R_i)$ is ground because $\sigma(R_i) = (\{U_i \mapsto D_i\}_{i \in [1..n]} \circ \{\sigma_i\}_{i \in [1..n]}) (R_i) = \{\sigma_i\}_{i \in [1..n]} (R_i)$ because \overline{U} are defined fresh.
 $(\{\sigma_i\}_{i \in [1..n]}) (R_i) = (\{\sigma_j\}_{j \in [1..i-1, i+1..n]} \circ \sigma_i) (R_i)$ by Corollary 1.
 $(\{\sigma_j\}_{j \in [1..i-1, i+1..n]}) (\sigma_i(R_i)) = \sigma_i(R_i)$ because $\sigma_i(R_i)$ is ground by *IH*. As a result $\sigma(R)$ is ground by definition of $merge_R$.
- 3) $\forall i \in [1..n]. \sigma(CR_i)$ is ground because $\sigma(CR_i) = (\{U_i \mapsto D_i\}_{i \in [1..n]} \circ \{\sigma_i\}_{i \in [1..n]}) (CR_i) = (\{\sigma_i\}_{i \in [1..n]}) (CR_i)$ because \overline{U} are defined fresh.
 $(\{\sigma_i\}_{i \in [1..n]}) (CR_i) = (\{\sigma_j\}_{j \in [1..i-1, i+1..n]} \circ \sigma_i) (CR_i)$ by Corollary 1.
 $(\{\sigma_j\}_{j \in [1..i-1, i+1..n]}) (\sigma_i(CR_i)) = \sigma_i(CR_i)$ because $\sigma_i(CR_i)$ is ground by *IH*.
 $\sigma(C.init(\overline{U}), \emptyset)$ is ground by $(*)$. As a result $\sigma(CR)$ is ground by definition of $merge_{CR}$.

The correspondence relation holds because:

- a) $C = \sigma(C)$
- b) $\forall i \in 1 \dots n. \Gamma \supseteq \sigma(R_i)$ because $\Gamma \supseteq \sigma_i(R_i)$ by *IH*, and from 2) $\sigma(R_i) = \sigma_i(R_i)$.
As a result $\Gamma \supseteq \sigma(R)$ by definition of $merge_R$.
- c) $\forall i \in 1 \dots n. CT$ satisfies $\sigma(CR_i)$ because CT satisfies $\sigma_i(CR_i)$ by *IH*, and from 3) $\sigma(CR_i) = \sigma_i(CR_i)$. CT satisfies $\sigma(C.init(\overline{U}), \emptyset)$ by $(**)$.
As a result CT satisfies $\sigma(CR_1) \dots \cup \sigma(CR_n) \cup \sigma(C.init(\overline{U}), \emptyset)$,
i.e., CT satisfies $\sigma(CR)$ by Lemma 4.

- Case T-UCAST with $\Gamma; CT \vdash (C)e : C$.

By inversion, $\Gamma; CT \vdash e : D$ and $D <: C$.

By *IH*, $e : T_e \mid S_e \mid R_e \mid CR_e$, where $solve(projExt(CT), S_e) = \sigma_e, \sigma_e(T_e), \sigma_e(R_e), \sigma_e(CR_e)$ are ground and the correspondence relation holds, i.e., $D = \sigma_e(T_e), \Gamma \supseteq \sigma_e(R_e), CT$ satisfies $\sigma_e(CR_e)$.

Let $\sigma = \sigma_e$, and $S = S_e \cup \{T_e <: C\}$.

Then $(C)e : C \mid S \mid R_e \mid CR_e$ holds by rule TC-UCAST.

σ solves S , because it solves S_e , and $\{T_e <: C\}$ as shown below:

- Since $\sigma = \sigma_e$ and σ_e solves S_e then σ solves S_e .
- Since $\{D <: C\}$ holds and $\sigma(\{T_e <: C\}) = \{D <: C\}$ then $\sigma(\{T_e <: C\})$ holds.

σ is ground solution because:

- 1) $\sigma(C)$ is ground because C is ground as a given class in CT
- 2) $\sigma(R_e)$ is ground because $\sigma_e(R_e)$ is ground by *IH* and $\sigma = \sigma_e$

B.2. Equivalence of Contextual and Co-Contextual FJ

3) $\sigma(CR_e)$ is ground because $\sigma_e(CR_e)$ is ground by IH and $\sigma = \sigma_e$

The correspondence relation $(C, \Gamma, CT) \triangleright (C, R_e, CR_e, \sigma)$ holds because:

a) $C = \sigma(C)$

b) $\Gamma \supseteq \sigma(R_e)$, because $\Gamma \supseteq \sigma_e(R_e)$ by IH and $\sigma = \sigma_e$

c) CT satisfies $\sigma(CR_e)$, because CT satisfies $\sigma_e(CR_e)$ by IH and $\sigma = \sigma_e$

The proof is symmetric for T-DCAST, and T-SCAST, as in the case of T-UCAST.

B. Equivalence of Contextual and Co-Contextual FJ

Definition 10 ($\text{CReqs}(\text{CR})$). $\text{CReqs}(\text{CR}) = \{T.\text{extends} : T' \mid (T.\text{extends} : T', \text{cond}) \in \text{CR}\} \cup \{T.\text{init}(\overline{T}) \mid (T.\text{init}(\overline{T}), \text{cond}) \in \text{CR}\} \cup \{T.f : T' \mid (T.f : T', \text{cond}) \in \text{CR}\} \cup \{T.m : \overline{T} \rightarrow T' \mid (T.m : \overline{T} \rightarrow T', \text{cond}) \in \text{CR}\}$

Definition 11 (Domain of Class Table Clause).

$$\text{domCl}(\text{CTcls}) = \begin{cases} (C.\text{extends}) & \text{if } (\text{CTcls}) = (C.\text{extends} = D) \\ (C.f) & \text{if } (\text{CTcls}) = (C.f : C_f) \\ (C.m) & \text{if } (\text{CTcls}) = (C.m : \overline{C} \rightarrow C_r) \\ (C.\text{init}) & \text{if } (\text{CTcls}) = (C.\text{init}(\overline{C})) \end{cases} \quad (\text{B.2})$$

Definition 12 (Domain of CT). $\text{dom}(\text{CT}) = \{\text{domCl}(\text{CTcls}) \mid \text{CTcls} \in \text{CT}\}$

Definition 13 (translate a class requirements to class table entries). *It is given a ground class requirement clause CReq.*

$$\text{translate}(\text{CReq}) = \begin{cases} (C.\text{extends} = D) & \text{if } (\text{CReq}) = (C.\text{extends} : D) \\ (C.f : C_f) & \text{if } (\text{CReq}) = (C.f : C_f) \\ (C.m : \overline{C} \rightarrow C_r) & \text{if } (\text{CReq}) = (C.m : \overline{C} \rightarrow C_r) \\ (C.\text{init}(\overline{C})) & \text{if } (\text{CReq}) = (C.\text{init}(\overline{C})) \end{cases} \quad (\text{B.3})$$

Definition 14 (translate a class table entry to a class requirement CReq). *It is given a class table clause CTcls.*

$$\text{translate}^*(\text{CTcls}) = \begin{cases} (C.\text{extends} : D) & \text{if } (\text{CTcls}) = (C.\text{extends} = D) \\ (C.f : C_f) & \text{if } (\text{CTcls}) = (C.f : C_f) \\ (C.m : \overline{C} \rightarrow C_r) & \text{if } (\text{CTcls}) = (C.m : \overline{C} \rightarrow C_r) \\ (C.\text{init}(\overline{C})) & \text{if } (\text{CTcls}) = (C.\text{init}(\overline{C})) \end{cases} \quad (\text{B.4})$$

Definition 15 (Clauses of supertypes of CReq).

$$\{(CReq, \text{CR})\}_{\ll} = \begin{cases} (T.\text{extends} : T') & \text{for } CReq = (T.\text{extends} : T') \\ \{(T.\text{init}(\overline{T}'))\} & \text{for } (T.\text{init}(\overline{T}')) \in \text{CReqs}(\text{CR}) \\ & \wedge CReq = (T.\text{init}(\overline{T})) \wedge \overline{T} <: \overline{T}' \\ \{(T'.f : T'_f)\} & \text{for } (T'.f : T'_f) \in \text{CReqs}(\text{CR}) \\ & \wedge CReq = (T.f : T_f) \wedge T <: T' \\ \{(T'.m : \overline{T}' \rightarrow T'_r)\} & \text{for } (T'.m : \overline{T}' \rightarrow T'_r) \in \text{CReqs}(\text{CR}) \\ & \wedge CReq = (T.m : \overline{T} \rightarrow T_r) \wedge T <: T' \end{cases} \quad (\text{B.5})$$

Definition 16 (Clauses of subtypes of CReq).

$$\{(CReq, CR)\}_{\gg} = \begin{cases} (T.\text{extends} : T') & \text{for } CReq = (T.\text{extends} : T') \\ \{(T.\text{init}(\overline{T}'))\} & \text{for } (T.\text{init}(\overline{T}')) \in CReqs(CR) \\ & \wedge CReq = (T.\text{init}(\overline{T})) \wedge \overline{T}' <: \overline{T} \\ \{(T'.f : T'_f)\} & \text{for } (T'.f : T'_f) \in CReqs(CR) \\ & \wedge CReq = (T.f : T_f) \wedge T' <: T \\ \{(T'.m : \overline{T}' \rightarrow T'_r)\} & \text{for } (T'.m : \overline{T}' \rightarrow T'_r) \in CReqs(CR) \\ & \wedge CReq = (T.m : \overline{T} \rightarrow T_r) \wedge T' <: T \end{cases} \quad (B.6)$$

Definition 17 (Clauses of superclasses of CTcls).

$$\{(CTcls, CT)\}_{\ll^*} = \begin{cases} (C.\text{extends} : D) & \text{for } CTcls = (C.\text{extends} = D) \\ \{(C.\text{init}(\overline{C}'))\} & \text{for } (C.\text{init}(\overline{C}')) \in CT \\ & \wedge CTcls = (C.\text{init}(\overline{C})) \wedge \overline{C} <: \overline{C}' \\ \{(D.f : D')\} & \text{for } (D.f : D') \in CT \\ & \wedge CTcls = (C.f : C') \wedge C <: D \\ \{(D.m : \overline{D} \rightarrow D_r)\} & \text{for } (D.m : \overline{D} \rightarrow D_r) \in CT \\ & \wedge CTcls = (C.m : \overline{C} \rightarrow C_r) \wedge C <: D \end{cases} \quad (B.7)$$

Definition 18 (Compatible class requirements). Given two class requirements $CReq, CReq'$, compatibility of two class requirements $CReq \sim CReq'$ is defined over all cases of clauses:

- $(T.\text{extends} : T_1) \sim (T'.\text{extends} : T_2)$ if $(T = T') \wedge (T_1 = T_2)$
- $T.\text{init}(\overline{T}) \sim (T'.\text{init}(\overline{T}'))$ if $(T = T')$
- $(T.f : T_f) \sim (T'.f : T'_f)$ if $(T <: T') \vee (T >: T')$
- $(T.m : \overline{T} \rightarrow T_r) \sim (T'.m : \overline{T}' \rightarrow T'_r)$ if $(T <: T') \vee (T >: T')$

Definition 19 (Compatibility between a class requirement and a class table clause). Given a class table clause $CTcls$, a class requirement $CReq$, and a ground solution σ , such that $\sigma(CReq)$ ground, compatibility $CReq \sim CReq'$ is defined over all cases of clauses:

- $\sigma(T.\text{extends} : T') \sim (C.\text{extends} = D)$ if $(\sigma(T) = C) \wedge (\sigma(T) = D)$
- $\sigma(T.\text{init}(\overline{T})) \sim (C.\text{init}(\overline{C}'))$ if $(\sigma(T) = C)$
- $\sigma(T.f : T_f) \sim (C.f : C')$ if $(\sigma(T) >: C) \vee (\sigma(T) <: C)$
- $\sigma(T.m : \overline{T} \rightarrow T_r) \sim (C.m : \overline{C} \rightarrow C')$ if $(\sigma(T) >: C) \vee (\sigma(T) <: C)$

Lemma 5 (Weakening for context). If $\Gamma \vdash t : T$, and $x \notin \text{dom}(\Gamma)$, then $\Gamma; x : C \vdash t : T$.

Proof. Straightforward induction on typing derivations. \square

Lemma 6 (Weakening for a single class requirement). Given CT , a class table clause $CTcls$, a class requirement $(CReq, \text{cond})$ and σ , such that $\sigma(CReq, \text{cond})$ is ground, if CT satisfies $\sigma(CReq, \text{cond})$ and $\forall CTcls' \in \{(CTcls, CT)\}_{\ll^*}$ such that $CTcls' \notin CT$, then $CT \cup CTcls$ satisfies $\sigma(CReq, \text{cond})$.

B. Equivalence of Contextual and Co-Contextual FJ

Proof. We proceed by case analysis on the definition of $CReq$.

- Case $CReq = (T.f : T')$. We consider $\sigma(T.f : T', cond) = (C.f : C', cond_g)$.

We have to show that $CT \cup CTcls$ satisfies $(C.f : C', cond_g)$.

It is given that CT satisfies $(C.f : C', cond_g)$, therefore by inversion $field(f, C, CT) = C'$ (rule S-FIELD). To show that the extended class table still satisfies the given class requirement, we distinguish the following cases on the definition of $CTcls$:

- 1) $CTcls = (D.g : D')$, such that $f \neq g$. Moreover, consider the class table $CT \cup (D.g : D')$. We know that since f is not the same as g :

$$(*) \text{ field}(f, C, CT) = \text{field}(f, C, CT \cup (D.g : D')) = C'.$$

As a result $CT \cup CTcls$ satisfies $(C.f : C', cond_g)$ by rule S-FIELD and $(*)$.

- 2) $CTcls = (A.f : A')$. Since by inversion $field(f, C, CT) = C'$, then there exists D , such that $C <: D$ and $(D.f : C') \in CT$. To proceed with the proof we distinguish two subcases:

- a) A and D belong to the same class hierarchy (subtyping relation).

$$A <: D$$

This case does not hold by the assumption that $\forall (CTcls') \in \{(A.f : A', CT)\} \ll^* \text{ such that } CTcls' \notin CT$, i.e., D is a supertype of A , and $D.f : C'$ is an existing clause of the class table.

$$A >: D$$

Since $C <: D$, then by transitivity we have $C <: A$. Thus the type of $C.f$ does not depend on the type of $A.f$, because by field lookup rule, the type of $C.f$ is defined by the first supertype we find starting from left to right; since $C <: D <: A$, then $D.f$ is considered to define the type of $C.f$. Moreover, consider the class table $CT \cup (A.f : A')$. We know that since $A >: D$, $A >: C$, from field lookup definition:

$$(*) \text{ field}(f, C, CT) = \text{field}(f, C, CT \cup (A.f : A')) = C'.$$

As a result $CT \cup CTcls$ satisfies $(C.f : C', cond_g)$ by $(*)$ and rule S-FIELD.

- b) A and D do not belong to the same class hierarchy (subtyping relation). We consider the class table $CT \cup (A.f : A')$. Since the field declaration for f of class A is unnecessary to define the type of $C.f$, because $C <: D$, and $D \not<: A$, $D \not>: A$, as a result $C \not<: A$, $C \not>: A$, then :

$$(*) \text{ field}(f, C, CT) = \text{field}(f, C, CT \cup (A.f : A')) = C'.$$

As a result $CT \cup CTcls$ satisfies $(C.f : C', cond_g)$ by $(*)$ and rule S-FIELD.

3) $CTcls$ is different from a field clause.

We consider the class table $CT \cup \text{translate}(CReq')$. We know that since $CReq'$ is different from field clause for class requirements:

$$(*) \text{ field}(f, C, CT) = \text{field}(f, C, CT \cup CTcls) = C'.$$

As a result $CT \cup CTcls$ satisfies $(C.f : C', \text{cond}_g)$ by $(*)$ and rule S-FIELD.

- $CReq = (T.m : \bar{U} \rightarrow U')$

□

Lemma 7 (Class Table Weakening). *Given CT , a class table clause $CTcls$, a set of class requirements CR , and a ground solution σ , such that $\sigma(CR)$ is ground, if CT satisfies $\sigma(CR)$ and $\forall CTcls' \in \{(CTcls, CT)\}^{\llast}$ such that $CTcls' \notin CT$, then $CT \cup CTcls$ satisfies $\sigma(CR)$.*

Proof. We proceed by mathematical induction on the set of class requirements CR .

Initial step: Show that the lemma holds for one single class requirement, i.e., $CR = \{(CReq, \text{cond})\}$. It is given a class table clause $CTcls$, $\sigma(CReq, \text{cond})$ is ground and CT satisfies $\sigma(CReq, \text{cond})$, then $CT \cup CTcls$ satisfies $\sigma(CReq, \text{cond})$ by Lemma 6.

Inductive step: We suppose that the lemma is true for a set of class requirements $CR = CR'$, i.e., $CT \cup CTcls$ satisfies $\sigma(CR')$, where $\sigma(CR')$ is ground.

We prove the lemma for $CR = (CReq, \text{cond}) \cup CR'$, i.e., $CT \cup CTcls$ satisfies $\sigma(CR)$.

Union of class requirements is realized by merge_{CR} function, i.e.,

$CR|_S = \text{merge}(CR', (CReq, \text{cond}))$. $\sigma(CReq, \text{cond})$ is ground from the initial step and $\sigma(CR')$ is ground from the inductive step, then σ solve S by Lemma 3. $CT \cup CTcls$ satisfies $\sigma(CReq, \text{cond})$ from the initial step, and $CT \cup CTcls$ satisfies $\sigma(CR')$ from the inductive step, as a result $CT \cup CTcls$ satisfies $\sigma(CReq, \text{cond}) \cup \sigma(CR')$, i.e., $CT \cup CTcls$ satisfies $\sigma(CR)$ by Lemma 4. □

Lemma 8 (Compatible clause in CT and not in CR). *Given $CT', CR', (CReq\emptyset)$, σ , such that $CR|_S = \text{merge}(CR', (CReq, \emptyset))$, σ solves S , and $\sigma(CR)$ is ground, if CT' satisfies $\sigma(CR')$, $\exists (CReq', \text{cond}) \in CR'$. $CReq \sim CReq'$, and $\exists CTcls \in CT'$. $\sigma(CReq) \sim CTcls$, then there exists a class table CT , such that CT satisfies $\sigma(CR)$.*

Proof. We proceed by case analyses on the definition of CReq.

- $CReq = (T.f : U)$, and $(D.f : D') \in CT'$ for some D , by assumption. We distinguish two cases regarding the subtyping relation between the CReq and the class table clause:

- 1) $D :> \sigma(T)$. Since $(D.f : D') \in CT$ is already a member of the class table, and D is supertype of $\sigma(T)$, then $\sigma(U) = D'$. We take $CT = CT'$. $\text{field}(f, \sigma(T), CT) = D'$, therefore CT satisfies $(\sigma(T).f : D', \emptyset)$ by rule S-FIELD, i.e., CT satisfies $(\sigma(T).f : D', \emptyset)$, and CT satisfies $\sigma(CR')$, as a result CT satisfies $\sigma(CR)$ by Lemma 4.

B. Equivalence of Contextual and Co-Contextual FJ

- 2) $D <: \sigma(T)$. We take $CT = CT' \cup \text{translate}(\sigma(T.f : U))$, then CT satisfies $\sigma(T.f : U, \emptyset)$ by construction and CT satisfies $\sigma(CR')$ by Class Table Weakening Lemma 7. As a result CT satisfies $\sigma(CR)$ by Lemma 4.

- $CReq = (T.m : \bar{U} \rightarrow U)$ Analogous to the case of field clause.

□

Lemma 9 (Compatible clause in CT and in CR). *Given CT', CR' , $(CReq, \emptyset)$, σ , such that $CR|_S = \text{merge}(CR', (CReq, \emptyset))$, σ solves S and $\sigma(CR)$ is ground, if CT' satisfies $\sigma(CR')$, $\exists (CReq', \text{cond}) \in CR'$. $CReq \sim CReq'$, $\exists CTcls \in CT'$. $\sigma(CReq) \sim CTcls$, then there exists a class table CT , such that CT satisfies $\sigma(CR)$.*

Proof. We proceed by case analyses on the definition of $CReq$.

$CReq = (T.f : U)$.

By assumption $(T'.f : T_f, \text{cond}') \in CR'$ for some T' , and $(D.f : D') \in CT'$, for some D , $\sigma(T') <: D$. To show that CT satisfies $\sigma(CR)$ we consider the case where $\sigma(\text{cond}') \Downarrow \text{true}$ ². $\sigma(\text{cond}) \Downarrow \text{true}$, i.e., all conditions in cond do hold. CT' satisfies $\sigma(CR')$, and $(T'.f : T_f, \text{cond}') \in CR'$, therefore CT' satisfies $\sigma(T'.f : T_f, \text{cond}')$, by inversion $\text{field}(f, T', CT') = D'$ (rule S-FIELD), where $\sigma(T_f) = D'$. We distinguish to cases with respect to the subtyping relation between D and $\sigma(T)$:

- 1) $D >: \sigma(T)$

$D >: \sigma(T)$, $D >: \sigma(T')$, let us consider $(*)$ $(\sigma(T). \text{extends} = D \in CT', (\sigma(T'). \text{extends} = D) \in CT')$ and $(D.f : D') \in CT'$. The class requirements we are interested in are $(T.f : U, \text{cond})$, $(T'.f : U', \text{cond}')$. After applying merging for the two requirements and remove for the two extend clauses the resulting valid requirements, that is the requirements where their conditions hold, are $(D.f : U, \text{cond}_t)$ and $(D.f : U', \text{cond}'_t)$ (for sake of brevity we omit the detailed steps and the non interesting requirements for us). Then after applying remove for the field clause results that $\sigma(U) = \sigma(U') = D'$. $\text{field}(f, \sigma(T'), CT') = D' = \sigma(U')$, $\text{field}(f, \sigma(T), CT') = D' = \sigma(U)$, therefore CT' satisfies $\sigma(T.f : U, \emptyset)$ by rule S-FIELD. We take $CT = CT'$. CT satisfies $\sigma(CR')$, and CT satisfies $\sigma(T.f : U, \emptyset)$, as a result CT satisfies $\sigma(CR)$ by Lemma 4.

- 2) $D <: \sigma(T)$

By transitivity $\sigma(T') <: \sigma(T)$. D is subtype of $\sigma(T)$ and $D.f$ is unnecessary to determine the type of $\sigma(T).f$ by field lookup rule. We take $CT = CT' \cup \text{translate}(\sigma(T.f : U))$. CT satisfies $\sigma(T.f : U, \emptyset)$ by class table construction, and $\sigma(T') <: D <: \sigma(T)$ then CT satisfies $\sigma(CR')$ by Class Table Weakening Lemma 7.

As a result CT satisfies $\sigma(CR)$ by Lemma 4.

$CReq = (T.m : \bar{U} \rightarrow U)$ Proof is analogous to case field clause.

□

²We do not consider when it is false because the requirement is not valid requirement and it is a case as in Lemma 8 and the proof follows the same

Lemma 10 (Add Clause Definition in CT). *Given a class table clause $CTcls$ declaration, a class table CT and a ground set of requirements CR , if $CTcls \notin CT$, and CT satisfies CR , then $CT \cup CTcls$ satisfies CR*

Proof. Tedious but straightforward. \square

Theorem 11 (Equivalence of expressions: \Leftarrow). *Given e, T, S, R, CR, Σ , if $e : T \mid S \mid R \mid CR$, $\text{solve}(\Sigma, S) = \sigma$, and σ is a ground solution, then there exists C, Γ, CT , such that*

$\Gamma; CT \vdash e : C, (C, \Gamma, CT) \triangleright \sigma(T, R, CR)$ and $\text{projExt}(CT) = \Sigma$.

We proceed by induction on the typing derivation.

- Case TC-VAR with $x : U \mid \emptyset \mid x : U \mid \emptyset$

Let σ be a ground solution, such that $\sigma(U)$ is ground by assumption.

By inversion, U is fresh, $S = \emptyset, R = \{x : U\}, CR = \emptyset$.

By IH, $\Gamma_x = \{x : \sigma(U)\}$

Let $\sigma(U) = C$, for some C we know it is ground.

Then $\Gamma; CT \vdash x : C$ by rule $T - Var$, and the correspondence relation holds:

- $\sigma(U) = C$
- We take $\Gamma = \Gamma_x$, and $\Gamma = \{x : C\} \supseteq \sigma(R) = \sigma(\{x : U\})$.
- We take $CT = \emptyset$, since $CR = \emptyset$, and $\sigma(CR) = \emptyset$, then CT satisfies $\sigma(CR)$

- Case TC-FIELD with $e.f_i : U \mid S \mid R_e \mid CR$

Let $S = S_e \cup S_f$, σ be a ground solution, such that $\text{solve}(S, \Sigma) = \sigma$, i.e., it solves S_e, S_f , and $\sigma(U), \sigma(R_e), \sigma(CR)$ are ground by assumption.

By inversion, $e : T_e \mid S_e \mid R_e \mid CR_e, \sigma(T_e), \sigma(R_e), \sigma(CR_e)$ are ground. $CR|_{S_f} = \text{merge}_{CR}(CR_e, (T_e.f_i : U, \emptyset))$, and U is fresh.

By IH, $\Gamma_e; CT_e \vdash e : C_e$, the correspondence relation holds, i.e., $C_e = \sigma(T_e), \Gamma_e \supseteq \sigma(R_e), CT_e$ satisfies $\sigma(CR_e)$. $\text{projExt}(CT_e) = \Sigma_e$

Let $C_i = \sigma(U)$, for some C_i we know is ground.

We consider three cases to construct the class table CT :

- (1) $\{(C_e.f : C_i, CT_e)\}_{\Leftarrow}^* = \emptyset$. Since no entry of class C_e or its superclasses exist for field f in the given class table CT_e , we add a new entry in the class table, i.e., $CT = CT_e \cup (C_e.f : C_i)$.
- (2) $\{(T_e'.f : U, CR_e)\}_{\Leftarrow} \cup \{(T_e'.f : U, CR_e)\}_{\gg} = \emptyset, (D.f : D') \in CT_e$ for some D, D' , then by Lemma 8 CT is constructed.
- (3) $(T'.f : T_f, \text{cond}') \in CR_e$, for some $T', \text{cond}', (D.f : D') \in CT_e$ for some $D, D', \sigma(T') <: D$, then by Lemma 9 CT is constructed.

B. Equivalence of Contextual and Co-Contextual FJ

From above we have that $field(f_i, C_e, CT) = C_i$, and no extends clauses are added to the class table CT_e , therefore $projExt(CT) = \Sigma_e = \Sigma$.

Then $\Gamma; CT \vdash e.f_i : C_i$ holds by rule T-FIELD, and the correspondence relation holds because:

- a) $\sigma(U) = C_i$
- b) We take $\Gamma = \Gamma_e$, and $\Gamma \supseteq \sigma(R_e)$ by IH.
- c) What it is left to be shown is that CT satisfies $\sigma(CR)$, we distinguish the following cases depending on the class table construction:
 - (1)' In addition to (1), $\sigma(T'_e.f : U) = \sigma(T'_e).f : \sigma(U) = C_e.f : C_i$, therefore CT satisfies $\sigma(T'_e.f : U, \emptyset)$ by construction of CT .
 CT_e satisfies $\sigma(CR_e)$ by IH, and $\{(C_e.f : C_i)\}_{\ll^*} \notin CT_e$ therefore CT satisfies $\sigma(CR_e)$ by Class Table Weakening Lemma 7.
 As a result CT satisfies $\sigma(CR_e) \cup \sigma(T'_e.f : U, \emptyset)$, i.e., CT satisfies $\sigma(CR)$ by Lemma 4.
 - (2)' In addition to (2), CT_e satisfies $\sigma(CR_e)$ by IH, then there is CT , CT satisfies $\sigma(CR)$ by Lemma 8.
 - (3)' In addition to (3), CT_e satisfies $\sigma(CR_e)$ by IH, then there is CT , CT satisfies $\sigma(CR)$ by Lemma 9.
- Case TC-INVK with $e.m(\bar{e}) : U \mid S \mid R \mid CR$.
 Let $S = S_e \cup \bar{S} \cup S_r \cup S_s \cup S_{cr} \cup \{\bar{T} <: \bar{U}\}$, and σ be a ground solution, such that it solves S , i.e., σ solves $S_e, \bar{S}, S_s, S_{cr}, \{\bar{T} <: \bar{U}\}$, and $\sigma(U'), \sigma(R), \sigma(CR)$ are ground.
 By inversion, $e : T_e \mid S_e \mid R_e \mid CR_e, \sigma(T_e), \sigma(R_e), \sigma(CR_e)$ are ground, $\bar{e} : \bar{T} \mid \bar{S} \mid \bar{R} \mid \bar{C}\bar{R}, \forall i \in [1..n]. \sigma(T_i), \sigma(R_i), \sigma(CR_i)$ are ground, $R|_{S_r} = merge_R(R_e, R_1, \dots, R_n)$, $CR'|_{S_s} = merge_{CR}(CR_e, CR_1, \dots, CR_n)$, $CR|_{S_{cr}} = merge_{CR}(CR', (T_e.m : \bar{U} \rightarrow U', \emptyset))$, and U', \bar{U} are fresh.
 By IH, $\Gamma_e; CT_e \vdash e : C_e$, the correspondence relation holds, with $C_e = \sigma(T_e)$, $\Gamma_e \supseteq \sigma(R_e)$, CT_e satisfies $\sigma(CR_e)$. $projExt(CT_e) = \Sigma_e$
 By IH, $\bar{\Gamma}; \bar{CT} \vdash \bar{e} : \bar{C}$, the correspondence relation holds, $\forall i \in [1..n]. C_i = \sigma(T'_i)$, $\Gamma_i \supseteq \sigma(R_i)$, CT_i satisfies $\sigma(CR_i)$. $projExt(CT_s) = \Sigma_s$, where:

$$\Gamma_s = \bigcup_{i \in [1..n]} \{\Gamma_i\} \quad CT_s = \bigcup_{i \in [1..n]} \{CT_i\}$$

- (*) $\{freshU(CT_e) \cap freshU(CT_s)\} = \emptyset$, and $\bigcap_{i \in [1..n]} \{freshU(CT_i)\} = \emptyset$, by Proposition 1. $\{CT_e \cup CT_s\}$ satisfies $\sigma(CR_e)$.
 $\forall i \in [1..n]. \{CT_e \cup CR_s\}$ satisfies $\sigma(CR_i)$ by Class Table Weakening Lemma 7, therefore $\{CT_e \cup CT_s\}$ satisfies $\sigma(CR_e) \cup \sigma(CR_1) \dots \cup \sigma(CR_n)$, i.e., $\{CT_e \cup CT_s\}$ satisfies $\sigma(CR')$ by Lemma 4.

B.2. Equivalence of Contextual and Co-Contextual FJ

Let $C = \sigma(U')$, $\bar{D} = \sigma(\bar{U})$ for some C, \bar{D} we know are ground. $\bar{C} <: \bar{D}$ holds because $\sigma(\{\bar{T} <: \bar{U}\})$ holds.

We consider three cases to construct the class table CT :

- (1) $\{(C_e.m : \bar{D} \rightarrow C, \{CT_e \cup CT_s\})\}_{\ll}^* = \emptyset$. Since no entry of class C_e or its superclasses exist for method m in the given class table $\{CT_e \cup CT_s\}$, we add a new entry in the class table, i.e., $CT = \{CT_e \cup CT_s\} \cup (C_e.m : \bar{D} \rightarrow C)$.
- (2) $\{(T_e.m : \bar{U} \rightarrow U', CR')\}_{\ll} \cup \{(T_e.m : \bar{U} \rightarrow U', CR')\}_{\gg} = \emptyset$, $(D.m : \bar{D} \rightarrow D') \in \{CT_e \cup CT_s\}$ for some D, \bar{D}, D' , then by Lemma 8 CT is constructed.
- (3) $(T'.m : \bar{T} \rightarrow T_r, cond') \in CR'$, for some $T', \bar{T}, T_r, cond'$, $(D.m : \bar{D} \rightarrow D') \in \{CT_e \cup CT_s\}$ for some D, \bar{D}, D' , $\sigma(T') <: D$, then by Lemma 9 CT is constructed.

From above we have that $mtype(m, C_e, CT) = \bar{D} \rightarrow C$, and no extends clauses are added to the class table $\{CT_e \cup CT_s\}$, therefore $projExt(CT) = \Sigma_e \cup \Sigma_s = \Sigma$.

Then $\Gamma; CT \vdash e.m(\bar{e}) : C$ holds by rule $T-INVK$, and the correspondence relation holds because:

- a) $C = \sigma(U)$
- b) We take $\Gamma = \Gamma_e \cup \Gamma_s$. $\Gamma \supseteq \sigma(R_e)$, because $\Gamma_e \supseteq \sigma(R_e)$ by *IH* and Context Weakening Lemma 5, $\Gamma \supseteq \sigma(R_1) \dots \Gamma \supseteq \sigma(R_n)$, because $\Gamma_i \supseteq R_i$ by *IH* and Context Weakening Lemma 5, therefore $\Gamma \supseteq \sigma(R)$ by definition of $merge_R$.
- c) What is left to be shown is that CT satisfies $\sigma(CR)$. We distinguish the following cases:
 - (1)' In addition to (1), $\sigma(T_e.m : \bar{U} \rightarrow U') = \sigma(T_e).m : \sigma(\bar{U}) \rightarrow \sigma(U') = C_e.m : \bar{D} \rightarrow C$ therefore CT satisfies $\sigma(T_e.m : \bar{U} \rightarrow U', \emptyset)$ by construction of CT . $\{CT_e \cup CT_s\}$ satisfies $\sigma(CR')$ by (*) therefore CT satisfies $\sigma(CR')$ by Class Table Weakening Lemma 7.
As a result CT satisfies $\sigma(CR') \cup \sigma(T_e.m : \bar{U} \rightarrow U, \emptyset)$, i.e., CT satisfies $\sigma(CR)$ by Lemma 4.
 - (2)' In addition (2), $\{CT_e \cup CT_s\}$ satisfies $\sigma(CR')$ by (*), then there is CT , CT satisfies $\sigma(CR)$ by Lemma 8.
 - (3)' In addition to (3), $\{CT_e \cup CT_s\}$ satisfies $\sigma(CR')$ by (*), then there is CT , CT satisfies $\sigma(CR)$ by Lemma 9.

- Case $TC-NEW$ with new $C(\bar{e}) : C \mid S \mid R \mid CR$

Let $S = \bar{S} \cup S_r \cup S_{cr} \cup \{\bar{T} <: \bar{U}\}$, σ be a ground solution, such that it solves S , i.e., σ solves $\bar{S}, S_r, S_{cr}, \{\bar{T} <: \bar{U}\}$, and $\sigma(C), \sigma(R), \sigma(CR)$ are ground.

By inversion, $\bar{e} : \bar{T} \mid \bar{S} \mid \bar{R} \mid \bar{C}\bar{R}, \forall i \in [1..n]$. $\sigma(T_i), \sigma(R_i), \sigma(CR_i)$ are ground, $R|_{S_r} = merge_R(R_1, \dots, R_n)$, $CR_s|_{S_s} = merge_{CR}(CR_1, \dots, CR_n)$, $CR|_{S_{cr}} = merge_{CR}(CR_s, (C.init(\bar{U}), \emptyset))$, and $\{U_i\}_{i \in [1..n]}$ are fresh.

B. Equivalence of Contextual and Co-Contextual FJ

By IH, $\bar{\Gamma}; \bar{CT} \vdash \bar{e} : \bar{C}$, the correspondence relation holds, $\forall i \in [1..n]$. $C_i = \sigma(T_i)$, $\Gamma_i \supseteq \sigma(R_i)$, CT_i satisfy $\sigma(CR_i)$. $\text{projExt}(CT_s) = \Sigma_s$, where:

$$\Gamma_s = \bigcup_{i \in [1..n]} \{\Gamma_i\} \quad CT_s = \bigcup_{i \in [1..n]} \{CT_i\}$$

(*) $\bigcap_{i \in [1..n]} \{\text{freshU}(CT_i)\} = \emptyset$, by Proposition 1.

$\forall i \in 1 \dots n$. CT_s satisfies $\sigma(CR_i)$ by Class Table Weakening Lemma 7, therefore CT_s satisfies $\sigma(CR_1) \dots \cup \sigma(CR_n)$, i.e., CT_s satisfies $\sigma(CR_s)$ by Lemma 4.

Let $\{U_i = D_i\}_{i \in [1..n]}$ for some C, \bar{D} we know are ground. $\bar{C} <: \bar{D}$ holds because $\sigma(\{\bar{T} <: \bar{U}\})$ holds.

We consider three cases to construct the class table CT :

- (1) $\{(C.\text{init}(\bar{D}), CT_s)\}_{\ll}^* = \emptyset$. Since no entry of class C exist for the constructor init in the given class table CT_s , we add a new entry in the class table, i.e., $CT = CT_s \cup (C.\text{init}(\bar{D}))$.
- (2) $\{(C.\text{init}(\sigma(\bar{U})), \sigma(CR_s))\}_{\ll} \cup \{(C.\text{init}(\sigma(\bar{U})), \sigma(CR_s))\}_{\gg} = \emptyset$, $(C.\text{init}(\bar{D}')) \in CT_s$, for some \bar{D}' , then by Lemma 8 CT is constructed.
- (3) $(C.\text{init}(\sigma(\bar{U}')), \sigma(\text{cond}')) \in \sigma(CR_s)$, for some \bar{U}', cond' , $(C.\text{init}(\bar{D}')) \in CT_s$, for some \bar{D}' , then by Lemma 9 CT is constructed.

From above we have that $\text{fields}(C, CT) = C.\text{init}(\bar{D})$, and no extends clauses are added to the class table CT_s , therefore $\text{projExt}(CT) = \Sigma_s = \Sigma$.

Then $\Gamma; CT \vdash C.\text{init}(\bar{e}) : C$ holds, the correspondence relation holds because:

- a) $C = \sigma(C)$
- b) We take $\Gamma = \Gamma_s$. $\Gamma_1 \supseteq \sigma(R_1) \dots \Gamma_n \supseteq \sigma(R_n)$ by IH, then by Context Weakening Lemma 5 $\Gamma \supseteq \sigma(R)$ by definition of merge_R .
- c) What is left to be shown is that CT satisfies $\sigma(CR)$. We distinguish the following cases:
 - (1)' In addition to (1), $\sigma(C.\text{init}(\bar{U})) = \sigma(C).\text{init}(\sigma(\bar{U})) = C.\text{init}(\bar{D})$ therefore CT satisfies $\sigma(C.\text{init}(\bar{U}), \emptyset)$ by construction of CT . CT_s satisfies $\sigma(CR_s)$ by (*), therefore CT satisfies $\sigma(CR_s)$ by Class Table Weakening Lemma 7. As a result CT satisfies $\sigma(CR_s) \cup \sigma(C.\text{init}(\bar{U}), \emptyset)$, i.e., CT satisfies $\sigma(CR)$ by Lemma 4.
 - (2)' In addition to (2), CT_s satisfies $\sigma(CR_s)$ by (*), then there is CT , CT satisfies $\sigma(CR)$ by Lemma 8.
 - (3)' In addition to (2), $\sigma(\bar{U}') <: \bar{D}'$, and CT_s satisfies $\sigma(CR_s)$ by (*), then there is CT , CT satisfies $\sigma(CR)$ by Lemma 9.

- Case TC-UCAST with $(C)e : C \mid S \mid R_e \mid CR_e$

Let $S = S_e \cup \{T'_e <: C\}$, σ be a ground solution, such that it solves S , i.e., σ solves S_e , $\{T'_e <: C\}$, and $\sigma(C)$, $\sigma(R_e)$, $\sigma(CR_e)$ are ground.

B.2. Equivalence of Contextual and Co-Contextual FJ

By inversion, $e : T_e \mid S_e \mid R_e \mid CR_e$, $\sigma(T_e)$, $\sigma(R_e)$, $\sigma(CR_e)$ are ground.

By IH, $\Gamma_e; CT_e \vdash e : D$, and the correspondence relation holds, i.e., $C_e = \sigma(T_e)$, and $\Gamma_e \supseteq \sigma(R_e)$, CT_e satisfies $\sigma(CR_e)$. $\text{projExt}(CT_e) = \Gamma_e$

$D <: C$ holds because $\sigma(\{T_e <: C\})$ holds.

Then $\Gamma; CT \vdash (C)e : C$ holds by rule T-UCAST, the correspondence relation holds because:

- a) $C = \sigma(C)$
- b) $\Gamma = \Gamma_e$, $\Gamma \supseteq \sigma(R_e)$ by IH
- c) $CT = CT_e$, CT satisfies $\sigma(CR_e)$ by IH

From above we have that no extends clauses are added to the class table CT_e , therefore $\text{projExt}(CT) = \Sigma_e = \Sigma$.

The proof is symmetric for T-DCAST, and T-SCAST, as in the case of T-UCAST.

□

Definition 20 (Correspondence relation for methods). *Given judgments $C; CT \vdash C_0 \ m(\overline{C} \ \overline{x})\{\text{return } e\} \text{ OK}$, $C_0 \ m(\overline{C} \ \overline{x})\{\text{return } e\} \text{ OK} \mid S \mid T \mid CR$, and $\text{solve}(\Sigma, S) = \sigma$, where $\text{projExt}(CT) = \Sigma$. The correspondence relation between CT and CR , written $(C, CT) \triangleright_m \sigma(T, CR)$, is defined as*

a) $C = \sigma(T)$

b) CT satisfies $\sigma(CR)$

Theorem 12 (Equivalence of methods: \Rightarrow). *Given m, C, CT , if $C; CT \vdash C_0 \ m(\overline{C} \ \overline{x})\{\text{return } e\} \text{ OK}$, then there exists S, T, CR, Σ, σ , where $\text{projExt}(CT) = \Sigma$ and $\text{solve}(\Sigma, S) = \sigma$, such that $C_0 \ m(\overline{C} \ \overline{x})\{\text{return } e_0\} \text{ OK} \mid S \mid T \mid CR$ holds, σ is a ground solution and $(C, CT) \triangleright_m \sigma(T, CR)$ holds.*

Proof. By induction on the typing judgment.

CASE T-METHOD with $C; CT \vdash C_0 \ m(\overline{C} \ \overline{x})\{\text{return } e\} \text{ OK}$.

By inversion, $\overline{x} : \overline{C}; \text{this} : C; CT \vdash e : E_0, \{E_0 <: C_0\}$, $\text{extends}(C, CT) = D$, i.e., $(C.\text{extends} = D) \in CT$ by rule EXTENDS, and if $\text{mtype}(m, D, CT) = \overline{D} \rightarrow D_0$, then $\overline{C} = \overline{D}; C_0 = D_0$.

By Theorem 10, $e_0 : T_e \mid S_e \mid R_e \mid CR_e$, where $\text{solve}(\text{projExt}(CT), S_e) = \sigma_e$, $\sigma_e(T_e)$, $\sigma_e(R_e)$, $\sigma_e(CR_e)$ are ground and the relation holds, i.e., $E_0 = \sigma_e(T_e)$, $\{\overline{x} : \overline{C}; \text{this} : C\} \supseteq \sigma_e(R_e)$, CT satisfies $\sigma_e(CR_e)$.

We define the set of constraints S' and the solution σ' depending on the occurrence of $\overline{x}, \text{this}$ in R_e , and U_c is fresh.

- If $\overline{x} \in \text{dom}(R_e)$ and $\text{this} \in \text{dom}(R_e)$, then $\{R_e(x_i) = U_i\}_{i \in [1..n]}$, $R_e(\text{this}) = U_c$, for \overline{U} fresh. We choose $S' = \{C_i = R_e(x_i)\}_{i \in [1..n]}; \{C = R_e(\text{this})\}$, $\sigma' = \{U_c \mapsto C\} \circ \{U_i \mapsto C_i\}_{i \in [1..n]}$.
- If $\overline{x} \in \text{dom}(R_e)$ and $\text{this} \notin \text{dom}(R_e)$, then $\{R_e(x_i) = U_i\}_{i \in [1..n]}$, for \overline{U} fresh. We choose $S' = \{C_i = R_e(x_i)\}_{i \in [1..n]}$, $\sigma' = \{U_c \mapsto C\} \circ \{U_i \mapsto C_i\}_{i \in [1..n]}$.
- If $\overline{x} \notin \text{dom}(R_e)$ and $\text{this} \in \text{dom}(R_e)$, then $R_e(\text{this}) = U_c$. We choose $S' = \{C = R_e(\text{this})\}$, $\sigma' = \{U_c \mapsto C\}$.
- If $\overline{x} \notin \text{dom}(R_e)$ and $\text{this} \notin \text{dom}(R_e)$. We choose $S' = \emptyset$, $\sigma' = \{U_c \mapsto C\}$.

In all the cases above we have $\{U_c \mapsto C\}$, regardless the occurrence of this in R_e , because U_c serves as a placeholder for the current class where the method m is declared as part of.

Let U_d be fresh, $R = R_e - \text{this} - \overline{x}$, $CR|_{S_{cr}} = \text{merge}_{CR}(CR_e, (U_c.\text{extends} : U_d, \emptyset), (U_d.m : \overline{C} \rightarrow C_0, \emptyset)_{\text{opt}})$, $S = S_e \cup \{T_e <: C_0\} \cup S_{cr} \cup S'$, $\sigma = \{U_d \mapsto D\} \circ \sigma' \circ \sigma_e$.

We show why R is \emptyset . The intuition behind it is that we know the actual types of the parameters since we have method declaration for m , and we know the actual type of this since it is given the current class C where method m is declared as part of. $\Gamma \supseteq \sigma(R_e)$ by

B.2. Equivalence of Contextual and Co-Contextual FJ

IH, i.e., all possible elements in R_e are \overline{X} , *this* and $\Gamma = \{\overline{x} : \overline{C}; \text{this} : C\} - \overline{x} - \text{this} = \emptyset$, therefore $R = R_e - \overline{x} - \text{this} = \emptyset$.

Then $C_0 \ m(\overline{C} \ \overline{x}) \ \{\text{return } e_0\} \ OK \mid S \mid U_c \mid CR$ holds by rule T-METHOD. σ solves S because it solves S_e , S' , S_{cr} , and $\{T_e <: C_0\}$ as shown below:

- $\text{solve}(\text{projExt}(CT), S_e) = \sigma_e$ and $\sigma = \{U_d \mapsto D\} \circ \sigma' \circ \sigma_e$ implies that σ solves S_e
- σ solves S' by Lemma 2.
- $\sigma_e(CR_e)$ is ground by Theorem 10.
 - (*) $\sigma(U_c.\text{extends} : U_d, \emptyset)$ is ground because $\sigma(U_c.\text{extends} : U_c) = (C.\text{extends} : D)$ and $C.\text{extends} : D$ is ground.
 - (**) $\sigma((U_d.m : \overline{C} \rightarrow C_0, \emptyset)_{opt})$ is ground because $\sigma(U_d.m : \overline{C} \rightarrow C_0) = (\sigma(U_d).m : \sigma(\overline{C}) \rightarrow \sigma(C_0)) = D.m : \overline{C} \rightarrow C_0$ and $D.m : \overline{C} \rightarrow C_0$ is ground.

CT satisfies $\sigma_e(CR_e)$ by Theorem 10.

(*) CT satisfies $\sigma(U_c.\text{extends} : U_d, \emptyset)$ because $(C.\text{extends} = D) \in CT$ hence by rule S-EXTENDS holds that CT satisfies $(C.\text{extends} : D, \emptyset)$, and $\sigma(U_c.\text{extends} : U_d) = (C.\text{extends} : D)$.

To show that CT satisfies $\sigma((U_d.m : \overline{C} \rightarrow C_0, \emptyset)_{opt})$ we distinguish the following cases:

- (*) if $\text{mtype}(m, D, CT) = \overline{D} \rightarrow D_0$ is true then the optional class requirement $(U_d.m : \overline{C} \rightarrow C_0)_{opt}$ is considered and $\overline{C} = \overline{D}, C_0 = D_0$, i.e., type of m declared in D is the same as the type of m declared in C . $\sigma(U_d.m : \overline{D} \rightarrow D_0) = D.m : \overline{D} \rightarrow D_0$ and $\text{mtype}(m, D, CT) = \overline{D} \rightarrow D_0 = \overline{C} \rightarrow C_0$, therefore by rule S-METHOD holds that CT satisfies $\sigma(U_d.m : \overline{C} \rightarrow C_0, \emptyset)$.
- (**) if $\text{mtype}(m, D, CT) = \overline{D} \rightarrow D_0$ is false, then the optional class requirement $(U_d.m : \overline{C} \rightarrow C_0)_{opt}$ is not considered. It is satisfiable by default since it is a not valid class requirement.

As a result σ solves S_{cr} by Lemma 3.

- Since $\{E_0 <: C_0\}$ holds and $\sigma(\{T_e <: C_0\}) = \{E_0 <: C_0\}$, then $\sigma(\{T_e <: C_0\})$ holds.

σ is ground solution because:

- 1) $\sigma(U)$ is ground because $\sigma(U) = C$ and C is ground.
- 2) $\sigma(CR_e)$ is ground because $\sigma(CR_e) = (\{U_d \mapsto D\} \circ \sigma' \circ \sigma_e)(CR_e) = (\{U_d \mapsto D\} \circ \sigma')(\sigma_e(CR_e)) = \sigma_e(CR_e)$ because $\sigma_e(CR_e)$ is ground by Theorem 10.
 $\sigma(U_c.\text{extends} : U_d, \emptyset)$ is ground by (*). $\sigma((U_d.m : \overline{C} \rightarrow C_0, \emptyset)_{opt})$ is ground by (**).
 As a result $\sigma(CR)$ is ground by definition of merge_{CR} .

The correspondence relation holds because:

- a) $C = \sigma(U_c)$

B. Equivalence of Contextual and Co-Contextual FJ

- b) CT satisfies $\sigma(CR_e)$ because CT satisfies $\sigma_e(CR_e)$ by Theorem 10 and from 3) $\sigma(CR_e) = \sigma_e(CR_e)$. CT satisfies $\sigma(U_c.\text{extends} : U_d, \emptyset)$ by (\star) . To show that CT satisfies $\sigma(CR)$, is left to scrutinize CT satisfies $\sigma((U_d.m : \bar{C} \rightarrow C_0, \emptyset)_{opt})$. We distinguish the following cases:

- if $mtype(m, D, CT) = \bar{D} \rightarrow D_0$ is true then CT satisfies $\sigma(U_d.m : \bar{C} \rightarrow C_0, \emptyset)$ by (\star') . As a result CT satisfies $\sigma(CR_e) \cup \sigma(U_c.\text{extends} : U_d, \emptyset) \cup \sigma(U_d.m : \bar{C} \rightarrow C_0, \emptyset)$, i.e., CT satisfies $\sigma(CR)$ by Lemma 4.
- if $mtype(m, D, CT) = \bar{D} \rightarrow D_0$ is false, then is not considered from (\star'') . As a result CT satisfies $\sigma(CR_e) \cup \sigma(U_c.\text{extends} : U_d, \emptyset)$, i.e., CT satisfies $\sigma(CR)$ by Lemma 4.

□

Theorem 13 (Equivalence of methods: \Leftarrow). *Given m, T, S, CR, Σ , if $C_0 m(\bar{C} \bar{x}) \{return e_0\} OK \mid S \mid T \mid CR$, $solve(\Sigma, S) = \sigma$, and σ is a ground solution, then there exists C, CT , such that $C; CT \vdash C_0 m(\bar{C} \bar{x}) \{return e\} OK$ holds, $(C, CT) \triangleright_m \sigma(T, CR)$ and $projExt(CT) = \Sigma$.*

Proof. By induction of the typing judgment, and case analysis of the class table construction.

Case TC-METHOD with $C_0 m(\bar{C} \bar{x}) \{return e\} OK \mid S \mid U_c \mid CR$.

Let $S = S_e \cup \{T_e <: C_0\} \cup S_c \cup S_{cr} \cup S_x$, σ be a ground solution, such that $solve(\Sigma, S) = \sigma$, i.e., σ solves $S_e, S_x, S_{cr}, S_c, \{T_e <: C_0\}$, and $\sigma(U_c), \sigma(CR)$ are ground.

By inversion, $e : T_e \mid S_e \mid R_e \mid CR_e$, $\sigma(T_e)$, $\sigma(R_e)$, $\sigma(CR_e)$ are ground. $CR'|_{S'} = merge_{CR}(CR_e, (U_c.\text{extends} : U_d, \emptyset))$ $CR|_{S_{cr}} = merge_{CR}(CR', (U_d.m : \bar{C} \rightarrow C_0, \emptyset)_{opt})$, where U_c, U_d are fresh, $R_e - this - \bar{x} = \emptyset$. σ solves S' by Lemma 2.

By Theorem 11, $\Gamma_e; CT_e \vdash e : E_0$, the correspondence relation holds, i.e., $E_0 = \sigma(T_e)$, $\Gamma_e \supseteq \sigma(R_e)$, CT_e satisfies $\sigma(CR_e)$. $\Gamma_e = \{\bar{x} : \bar{C}; this : C\}$, because $R_e - this - \bar{x} = \emptyset$ and $\Gamma_e \supseteq \sigma(CR_e)$. $projExt(CT_e) = \Sigma_e$

Let $C = \sigma(U_c)$, $D = \sigma(U_d)$ for some C, D we know are ground. $E_0 <: C_0$ holds because $\sigma(T_e <: C_0)$ holds.

Context empty because $\Gamma_e - \{\bar{x} : \bar{C}; this : C\} = \emptyset$.

We proceed by construction of the class table in steps.

First we consider three cases to construct the class table CT' with respect to the requirement for *extends*:

- (1) **clause not in class table.** $(C.\text{extend} = D) \notin CT_e$, then $CT' = CT_e; (C.\text{extends} = D)$.
- (2) **clause in class table, but not in class requirements.** $(C.\text{extends} = D) \in CT_e$, and $(U_c.\text{extends} : U_d, \emptyset) \notin CR_e$, then $CT' = CT_e$.
- (3) **clause in class table and class requirements.** $(C.\text{extends} = D) \in CT_e$, and $(U_c.\text{extends} : U_d, \emptyset) \in CR_e$ is not a valid case, because U_c is defined fresh and in CR_e

we do not have existing requirements regarding U_c for extend, because in method body we can have recursive method call or field access and not in extends, i.e., *this* can invoke the method itself or other methods and fields but not extends.

From above and by rule EXTENDS we have that $extends(C, CT') = D$, an extends is added to the class table CT_e , therefore $projExt(CT') = \Sigma_e \cup (C, D) = \Sigma'$

Second we consider three cases to construct the class table CT' with respect to the requirement for method m :

- (4) **clauses of superclasses not in class table.** $\{(D.m : \bar{D} \rightarrow D_0, CT')\}_{\ll}^* = \emptyset$, then $CT = CT'$
- (5) **compatible clauses in class table, but not in class requirements.** $\{(D.m : \bar{C} \rightarrow C', CR_e)\}_{\ll} \cup \{(D.m : \bar{D} \rightarrow D', CR_e)\}_{\gg} = \emptyset$, $(D'.m : \bar{D}' \rightarrow D_r) \in CT'$ for some D', \bar{D}', D_r , then by Lemma 8 CT' is constructed.
- (6) **compatible clauses in class table, and in class requirements.** $(D'.m : \bar{D}' \rightarrow D_r) \in CT'$ for some D', \bar{D}', D_r , and $(U_d.m \bar{C} \rightarrow C_0, \emptyset) \in CR_e$ is not a valid case because U_d is defined fresh and $U_d \neq R_e(this)$, i.e., it is possible to have in the body of m *this.m*, but it is impossible to have recursive call of m invoked by U_d , as it is defined fresh and different type than *this*.

From above we have that if $mtype(m, D, CT) = \bar{D} \rightarrow D_0$ then $\bar{C} = \bar{D}; C_0 = D_0$, no extends clauses are added to the class table CT' , therefore $projExt(CT) = \Sigma' = \Sigma$

Then $C; CT \vdash C_0 \ m(\bar{C} \ \bar{x})\{return \ e\}$ OK holds by rule T-METHOD, the correspondence relation holds because:

- a) $C = \sigma(U_c)$
- b) What is left to be shown is that CT satisfies $\sigma(CR)$, first we start by showing CT' satisfies $\sigma(CR')$ and we distinguish the following cases:
 - (1)' In addition to (1) $\sigma(U_c.extends : U_d) = \sigma(U_c).extends : \sigma(U_d) = C.extends : D$ therefore CT' satisfies $\sigma(U_c.extends : U_d, \emptyset)$ by construction. CT_e satisfies $\sigma(CR_e)$ by Theorem 11, and $\sigma(U_c.extends : U_d) \notin CT_e$, therefore CT' satisfies $\sigma(CR_e)$ by Class Table Weakening Lemma 7.
As a result CT' satisfies $\sigma(CR_e) \cup \sigma(U_c.extends : U_d, \emptyset)$, i.e., CT' satisfies $\sigma(CR)$ by Lemma 4.
 - (2)' In addition to (2), CT' satisfies $(C.extends : D, \emptyset)$ by rule S-EXTEND, and $(C.extends : D) = \sigma(U_c.extends : U_d)$, therefore CT' satisfies $(U_c.extends : U_d, \emptyset)$. CT' satisfies $\sigma(CR_e)$ by Theorem 11. As a result CT' satisfies $\sigma(CR_e) \cup \sigma(U_c.extends : U_d, \emptyset)$, i.e., CT' satisfies $\sigma(CR)$ by Lemma 4.

Second we show that CT satisfies $\sigma(CR)$, and we distinguish the following cases:

B. Equivalence of Contextual and Co-Contextual FJ

- (3)' In addition to (4), the class requirement $(U_d.m : \overline{C} \rightarrow C_0)_{opt}$ is not considered since it is an optional requirement, therefore $CR = CR'$, CT' satisfies $\sigma(CR')$. As a result CT satisfies $\sigma(CR)$.
- (4)' In addition to (5), CT satisfies $\sigma(CR)$ by Lemma 8.

Method declaration consist in adding method clause m in CT , whether it is already member of the CT or not. Also, adding the method m in CT does not affect the satisfaction of the class requirements. We are interested that the clause m with its actual type is part of class table. Namely resulting class table CT_r , such that $(C.m : \overline{C} \rightarrow C_0) \in CT_r$, CT_r satisfies $\sigma(CR)$.

Lastly we show that CT_r satisfies $\sigma(CR)$ and we distinguish the following cases:

- $(C.m : \overline{C} \rightarrow C_0) \notin CT$ then we add declaration in the class table, i.e., $CT_r = CT \cup (C.m : \overline{C} \rightarrow C_0)$ and CT_r satisfies $\sigma(CR)$ by Lemma 10.
- $C.m \in \text{dom}(CT)$ then $CT_r = CT$. Hence, CT_r satisfies $\sigma(CR)$.

□

Definition 21 (Correspondence relation for classes). *Given $CT \vdash$ class C extends $D \{\overline{C} \ \overline{f}; K \ \overline{M}\} OK$ and class C extends $D \{\overline{C} \ \overline{f}; K \ \overline{M}\} OK \mid S \mid CR$, and $\text{solve}(\Sigma, S) = \sigma$, where $\text{projExt}(CT) = \Sigma$. The correspondence relation between CT and CR , written $(CT) \triangleright_c \sigma(CR)$, is defined as:*

a) CT satisfies $\sigma(CR)$

Theorem 14 (Equivalence of classes: \Rightarrow). *Given C , CT , if $CT \vdash$ class C extends $D \{\overline{C} \ \overline{f}; K \ \overline{M}\} OK$, then there exists S , CR , Σ , σ , where $\text{projExt}(CT) = \Sigma$ and $\text{solve}(\Sigma, S) = \sigma$, such that class C extends $D \{\overline{C} \ \overline{f}; K \ \overline{M}\} OK \mid S \mid CR$ holds, σ is a ground solution and $(CT) \triangleright_c \sigma(CR)$ holds.*

Proof. By induction on the typing judgment.

CASE T-CLASS with $CT \vdash$ class C extends $D \{\overline{C} \ \overline{f}; K \ \overline{M}\} OK$.

By inversion, $K = C(\overline{D'} \ \overline{g}, \overline{C'} \ \overline{f})\{\text{super}(\overline{g}); \text{this}.\overline{f} = \overline{f}\}$, i.e., the constructor initializes all fields of $\text{fields}(D, CT) = D.\text{init}(\overline{D'})$, and $C; CT \vdash \overline{M} OK$.

By Theorem 12, $\overline{M} OK \mid \overline{S} \mid \overline{U} \mid \overline{CR}$, $\forall i \in 1 \dots n$. $\text{solve}(\text{projExt}(CT), S_i) = \sigma'_i$, $\sigma_i = \{U_i \mapsto C\} \circ \sigma'_i$, $\sigma_i(U_i)$, $\sigma_i(CR_i)$ are ground and the correspondence relation holds, i.e., $C = \sigma_i(U_i)$, CT satisfies $\sigma_i(CR_i)$.

Let $CR|_{S_{cr}} = \text{merge}_{CR}(CR_1, \dots, CR_n, D.\text{init}(\overline{D'}))$, $S = \overline{S} \cup S_{cr} \cup \{U_i = C\}_{i \in [1..n]} \cup \{C_i = D'_i\}_{i \in 1..k} \cup \{C_i = C'_i\}_{i \in k..n}$, where $k = |\overline{D'}|$, $n = |\overline{C}|$, $n - k = |\overline{C'}|$, and $\sigma = \{\sigma_i\}_{i \in [1..n]}$.

Then class C extends $D \{\overline{C} \ \overline{f}; K \ \overline{M}\} OK \mid S \mid CR$ holds by rule TC-CLASS.

σ solves \overline{S} , S_{cr} , $\{U_i = C\}_{i \in [1..n]}$ and $\{C_i = D'_i\}_{i \in 1..k} \cup \{C_i = C'_i\}_{i \in k..n}$ as shown below:

- σ solves \overline{S} because $\sigma = \{\{U_i \mapsto C\} \circ \sigma'_i\}_{i \in [1..n]}$, and $\forall i \in [1..n]$. $\text{solve}(\text{projExt}(CT), S_i) = \sigma_i$.
- $\forall i \in [1..n]$. $\sigma_i(CR_i)$ are ground by Theorem 12.
 - (*) $\sigma(D.\text{init}(\overline{D'}))$ is ground because $(D.\text{init}(\overline{D'}))$ is ground.
 - $\forall i \in [1..n]$. CT satisfies $\sigma_i(CR_i)$ by Theorem 12.
 - (**) CT satisfies $\sigma(D.\text{init}(\overline{D}), \emptyset)$ because $\text{fields}(D, CT) = D.\text{init}(\overline{D'})$ hence by rule S-CONSTRUCTOR holds that CT satisfies $\sigma(D.\text{init}(\overline{D'}), \emptyset)$. As a result σ solves S_{cr} by Lemma 3.
- σ solves $\{U_i = C\}_{i \in [1..n]}$ because $\sigma = \{\{U_i \mapsto C\} \circ \sigma'_i\}_{i \in [1..n]}$.
- $\{C_i = D'_i\}_{i \in 1..k} \cup \{C_i = C'_i\}_{i \in k..n}$ holds because K initializes all fields of class C as it is given by inversion.

σ is ground solution because:

- 1) $\forall i \in 1 \dots n$. $\sigma(CR_i)$ is ground because $\sigma(CR_i) = (\{\sigma_j\}_{j \in [1..i-1, i+1..n]} \circ \sigma_i)(CR_i)$ by Corollary 1.
 - $(\{\sigma_j\}_{j \in [1..i-1, i+1..n]})(\sigma_i(CR_i)) = \sigma_i(CR_i)$ because $\sigma_i(CR_i)$ is ground by Theorem 12.
 - $\sigma(D.\text{init}(\overline{D'}))$ is ground by (*). As a result $\sigma(CR)$ is ground by definition of merge_{CR}

B. Equivalence of Contextual and Co-Contextual FJ

The correspondence relation holds because:

- a) $\forall i \in 1 \dots n$. CT satisfies $\sigma(CR_i)$ because CT satisfies $\sigma_i(CR_i)$ by Theorem 12, and from 1) $\sigma(CR_i) = \sigma_i(CR_i)$. CT satisfies $\sigma(D.init(\overline{D}), \emptyset)$ by (**). As a result CT satisfies $\sigma(CR_1) \dots \cup \sigma(CR_n) \cup \sigma(D.init(\overline{D}'))$, i.e., CT satisfies $\sigma(CR)$ by Lemma 4.

□

Theorem 15 (Equivalence of classes: \Leftarrow). *Given C , CR , Σ , if class C extends D $\{\overline{C} \ \overline{f}; K \ \overline{M}\} OK \mid S \mid CR$, solve(Σ, S) = σ , and σ is a ground solution, then there exists CT , such that $CT \vdash$ class C extends D $\{\overline{C} \ \overline{f}; K \ \overline{M}\} OK$ holds, $(CT) \triangleright_c \sigma(CR)$ holds and $\text{projExt}(CT) = \Sigma$.*

Proof. By induction on the typing judgment.

Case TC-CLASS with class C extends D $\{\overline{C} \ \overline{f}; K \ \overline{M}\} OK \mid S \mid CR$.

Let $S = \overline{S} \cup S_{cr} \cup \{U_i = C\}_{i \in [1..n]} \cup \{C_i = D'_i\}_{i \in [1..k]} \cup \{C_i = C'_i\}_{i \in [k..n]}$, where $k = |\overline{D}'|$, $n = |\overline{C}|$, $n - k = |\overline{C}'|$, σ be a ground solution, such that it solves S and $\sigma(CR)$ is ground.

By inversion, $\overline{M} OK \mid \overline{S} \mid \overline{U} \mid \overline{C}R$, $\forall i \in 1 \dots n$. $\sigma(U_i)$, $\sigma(CR_i)$ are ground.

$\text{merge}_{CR}(CR_1, \dots, CR_n) = CR'|_{S_c}$, $\text{merge}_{CR}((D.init(\overline{D}')), CR') = CR'|_{S_{cr}}$.

Let $\forall i \in i \dots n$. $\sigma(U_i) = C$ for C we know it is ground.

By Theorem 13 $C; \overline{C}T \vdash \overline{M} OK$, the correspondence relation holds, $\forall i \in 1 \dots n$. $C = \sigma(U_i)$, CT_i satisfies $\sigma(CR_i)$. $\text{projExt}(CT') = \Sigma'$, where $CT' = \bigcup_{i \in [1..n]} \{CT_i\}$.

(*) $\bigcap_{i \in [1..n]} \{\text{freshU}(CR_i)\} = \emptyset$ by Proposition 1. $\forall i \in 1 \dots n$. CT' satisfies $\sigma(CR_i)$ by Class Table Weakening Lemma, therefore CT' satisfies $\sigma(CR')$ by Lemma 4.

The constructor K initializes all fields of class C , i.e., $K = C(\overline{D}' \ \overline{g}, \overline{C}' \ \overline{f})\{\text{super}(\overline{g}); \text{this}.\overline{f} = \overline{f}\}$, because σ solves $\{C_i = D'_i\}_{i \in [1..k]} \cup \{C_i = C'_i\}_{i \in [k..n]}$.

We consider three cases to construct the class table CT :

- (1) $\{(D.init(\overline{D}'), CT')\}_{\Leftarrow} = \emptyset$. Since no entry of class D exist for the constructor $init$ in the given class table CT' , we add a new entry in the class table, i.e., $CT = CT' \cup (D.init(\overline{D}'))$.
- (2) $\{(D.init(\overline{D}'), \sigma(CR'))\}_{\Leftarrow} \cup \{(D.init(\overline{D}'), \sigma(CR'))\}_{\gg} = \emptyset$, $(D.init(\overline{D}'')) \in CT'$, for some \overline{D}'' , then by Lemma 8 CT is constructed.
- (3) $(D.init(\overline{A})\text{cond}') \in \sigma(CR')$, for some $\overline{A}, \text{cond}'$, $(D.init(\overline{D}'')) \in CT'$, for some \overline{D}'' , then by Lemma 9 CT is constructed.

From above we have that $\text{fields}(D, CT) = D.init(\overline{D}')$, no extends clauses are added to the class table CT' , therefore $\text{projExt}(CT) = \Sigma' = \Sigma$

Then $CT \vdash$ class C extends D $\{\overline{C} \ \overline{f}; K \ \overline{M}\} OK \mid \overline{S}$ holds by rule T-CLASS.

The correspondence relation holds because:

- a) We have to show is that CT satisfies $\sigma(CR)$, and we distinguish the following cases:

- (1)' In addition to (1) CT satisfies $\sigma(D.init(\overline{D'}))$ by construction, CT' satisfies $\sigma(CR')$ by (*), therefore CT satisfies $\sigma(CR')$ by Class Table Weakening Lemma 7. As a result CT satisfies $\sigma(D.init(\overline{D'})) \cup \sigma(CR')$, i.e., CT satisfies $\sigma(CR)$ by definition of $merge_{CR}$.
- (2)' In addition to (2), CT' satisfies $\sigma(CR')$ by (*), then there is CT , CT satisfies $\sigma(CR)$ by Lemma 8.
- (3)' In addition to (3), CT' satisfies $\sigma(CR')$ by (*), then there is CT , CT satisfies $\sigma(CR)$, by Lemma 9.

Class declaration consists in adding the class C with all of its fields, methods, constructor and extend clauses in the class table, whether they are already member of the CT or not. Also, adding these clauses does not affect the satisfaction of the class requirements. Namely resulting class table CT_r , such that $C.extends = D \in CT_r$, $K \in CT_r$, $\{C.f_i : C_i\}_{i \in [1..n]} \in CT_r$, $\overline{M} \in CT_r$, CT_r satisfies $\sigma(CR)$. We distinguish the following cases:

- $(C.extends = D) \notin CT$, or $(C.init(\overline{C})) \notin CT$, or $\{C.f_i : C_i\}_{i \in [1..n]} \notin CT$, or $\{C.m_i : \overline{C} \rightarrow C_0\}_{i \in [1..n]} \notin CT$ then $CT_r = CT \cup (C.extends = D); (C.init(\overline{C})) \cup \{C.f_i : C_i\}_{i \in [1..n]} \cup \{C.m_i : \overline{C} \rightarrow C_0\}_{i \in [1..n]}$, and CT_r satisfies $\sigma(CR)$ by Lemma 10.
- $\forall CTcls \in \{(C.extends = D) \cup (C.init(\overline{C})) \cup \{C.f_i : C_i\}_{i \in [1..n]} \cup \{C.m_i : \overline{C} \rightarrow C_0\}_{i \in [1..n]}\}$ such that $domCl(CTcls) \in dom(CT)$ then $CT_r = CT$. Hence, CT_r satisfies $\sigma(CR)$.

□

B. Equivalence of Contextual and Co-Contextual FJ

Lemma 11. *Given a complete class table CT constructed from all possible class declarations \bar{L} , a set of requirements CR , $\biguplus_{L' \in \bar{L}}(\text{removeMs}(CR, L') \uplus \text{removeFs}(CR, L') \uplus \text{removeCtor}(CR, L') \uplus \text{removeExt}(CR, L')) = CR'|_S$, a substitution σ , such that $\sigma(CR)$ is ground, and CT satisfies $\sigma(CR)$. Then σ solves S .*

Proof. By the definitions of remove for different clauses, $S = S_c \cup S_e \cup S_k \cup S_f \cup S_m$. Let us consider constrains generated from field remove S_f . Suppose there exist $f \in \text{dom}(CR)$ such that $(T.f : T', \text{cond}) \in CR$ and $\sigma(\text{cond})$ hold.

Let $\sigma(T) = C$ and $\sigma(T') = C_f$, since $\sigma(CR)$ ground, C, C_f are ground.

Since CT satisfies $\sigma(CR)$, by inversion $\text{field}(f, C, CT) = C_f$, i.e, exist $D > C$ such that $D.f : \sigma(T') \in CT$. We distinguish two cases when f is declared in C or in one of its superclasses D :

- 1) $D = C$. By rule $S\text{-FIELD}$; $C.f : C_f \in CT$. We apply remove for field clause $C.f : C_f$. By definition of removeF the correspondent requirement is $(T.f : T', \text{cond}) \in CR$ and the new constraint generated is $S_f = (T' = C_f \text{ if } T = C)$. This constraint is solved, because the condition holds and $\sigma(T') = C_f$.
- 2) $D > C$. Then there exist C extends $D \in CT$ and $D.f : C_f \in CT$. In this case we have to apply remove for extends and field clauses. First, we apply remove of extends. By definition of removeExt the requirement under scrutiny is duplicated, i.e., $(T.f : T', \text{cond} \cup T \neq C), (D.f : T', \text{cond} \cup T = C)$. Second we apply remove of field f . By definition of removeFs the generated constrains are $S_f = \{(T' = C_f \text{ if } T = D), (T' = C_f) \text{ if } D = D\}$. The first constraint is not valid because the condition does not hold ($\sigma(T) \neq D$), therefore is not considered. the second constraint is solved because the condition holds and $\sigma(T') = C_f$.

The same procedure we follow for extends, constructors and methods clauses. \square

Lemma 12 (Class requirements empty). *Given a complete class table CT constructed from all possible class declarations \bar{L} , a set of requirements CR , $\biguplus_{L' \in \bar{L}}(\text{removeMs}(CR, L') \uplus \text{removeFs}(CR, L') \uplus \text{removeCtor}(CR, L') \uplus \text{removeExt}(CR, L')) = CR'|_S$, and a substitution σ , such that $\sigma(CR)$ is ground, σ solves S , we have that if CT satisfies $\sigma(CR)$, then and $\sigma(CR') = \emptyset$.*

Proof. By contradiction.

By assumption $\sigma(CR') \neq \emptyset$, and $CR' = \{(CReq, \text{cond}) \mid \exists (T \neq T') \in \text{cond}\}$. From this, follows $\forall (CReq, \text{cond}) \in CR'$. cond holds, i.e., all conditions of cond do hold. This is derived after performing remove, we already know the exact types for classes and their extends, constructor, fields, method clauses. Therefore from remove we add inequalities to invalidate requirements for which we know their exact types, as result exist one their conditions that does not hold. Since by assumption the set is not empty then all conditions hold. For sake of brevity we consider only the conditions that are added after performing remove, because are the ones we are interested in.

First we consider the extend clauses in the requirement set. All conditions of the requirements corresponding extends clause do hold. Let us consider $\exists (T.\text{extends}T', \text{cond}) \in$

CR' . $\forall(T \neq C) \in \text{cond}$. $\sigma(T) \neq C$ holds. By definition it is given that $\sigma(CR)$ is ground, namely $\sigma(T) = C'$, $\sigma(T') = D'$, such that C', D' are ground. Since all the inequalities in cond hold, this means that in the class table was not added any *extends clause*, such that $(C'.\text{extends} = D') \notin CT$. Therefore CT satisfies $\sigma(CR)$ does not hold.

This strategy of proof is used for constructor since from the definition of *removeCtor* only inequality conditions are added, and not considered while removing extends clause.

Second we consider field clauses. From the definition of *removeFs* and *removeExt* for every field clause we have a duplicated requirement corresponding to the parents type. All conditions of the requirements corresponding field clause do hold. By definition it is given that $\sigma(CR)$ is ground, namely $\sigma(T) = C'$, $\sigma(T_f) = C_f$, such that C', C_f are ground. Let us consider $(C'.\text{extends} = D) \in CT$, and $\exists(T.f : T_f, \text{cond} \cup T \neq C')$, $(D.f : T_f, \text{cond}' \cup T = C') \in CR'$ such that $\forall(T \neq C), (T = C) \in \text{cond} \cup \text{cond}' \cup T = C' \cup T \neq C' (\sigma(T) \neq C), (\sigma(T) = C)$ hold. Since all the conditions in $\text{cond} \cup \text{cond}'$ hold, this means that in the class table was not added any *field clause*, such that f is declared in C' or in its parents, i.e., $\forall C''$ such that $C' <: C''$, then $(C''.f : C_f) \notin CT$.

Therefore CT satisfies $\sigma(CR)$ does not hold.

The same strategy of proving is used for methods. In contrast for the optional methods regardless all the conditions might hold they are removed in any case, because they are optional. The lack of inequality conditions that do not hold, only shows the given method is declared in a class of the class table but not in its parents.

□

Theorem 16 (Equivalence for programs: \Rightarrow). *Given \bar{L} , if $\bar{L} OK$, then there exists S, Σ, σ , where $\text{projExt}(\bar{L}) = \Sigma$ and $\text{solve}(\Sigma, S) = \sigma$, such that $\bar{L} OK \mid S$ holds and σ ground solution.*

Proof. By induction on the typing judgment.

Case T-PROGRAM with $\bar{C} \bar{L} OK$.

By inversion, *Class table construction* CT is

$CT = \bigcup_{L' \in \bar{L}} (\text{addExt}(L') \cup \text{addCtor}(L') \cup \text{addFs}(L') \cup \text{addMs}(L'))$ and $CT \mid \bar{L} OK$.

By Theorem 14, $\bar{L} OK \mid \bar{S} \mid \bar{C}\bar{R}$, $\forall i \in 1..n$. $\text{solve}(\text{projExt}(CT), S_i) = \sigma_i$, $\sigma_i(CR_i)$ is ground and the correspondence relation holds, i.e., CT satisfies $\sigma_i(CR_i)$.

Let $CR_{|S_{cr}} = \text{merge}_{CR}(CR_1, \dots, CR_n)$, $\biguplus_{L' \in \bar{L}} (\text{removeMs}(CR, L') \uplus \text{removeFs}(CR, L') \uplus \text{removeCtor}(CR, L') \uplus \text{removeExt}(CR, L')) = CR_f|_{S_r}$, $S = \bar{S} \cup S_{cr} \cup S_r$, and $\sigma = \{\sigma_i\}_{i \in [1..n]}$. From the Lemma 12 we have $\sigma(CR_f) = \emptyset$.

Then $\bar{L} OK \mid S$ holds by rule TC-PROGRAM.

σ solves \bar{S} , and S_{cr} as shown below:

- σ solves \bar{S} because $\sigma = \{\sigma_i\}_{i \in [1..n]}$.
- $\forall i \in [1..n]$. $\sigma_i(CR)_i$ are ground by Theorem 14.
- $\forall i \in [1..n]$. CT satisfies $\sigma_i(CR_i)$ by Theorem 14.

B. Equivalence of Contextual and Co-Contextual FJ

As a result σ solves S_{cr} by Lemma 3.

- $\sigma(CR)$ is ground and CT satisfies $\sigma(CR)$ by Theorem 14, and given the class table CT , then σ solves S_r by Lemma 11.

σ is ground solution because:

- 1) $\forall i \in [1..n]. \sigma(\text{projExt}(CT), CR_i)$ is ground because $\sigma(CR_i) = (\{\sigma_i\}_{i \in [1..n]})$
 $(CR_i) = (\{\sigma_j\}_{j \in [1..i-1, i+1..n]} \circ \sigma_i)(CR_i)$ by Corollary 1.
 $(\{\sigma_j\}_{j \in [1..i-1, i+1..n]})(\sigma_i(CR_i)) = \sigma_i(CR_i)$ because $\sigma_i(CR_i)$ is ground by Theorem 14.
As a result $\sigma(CR)$ is ground by definition of merge_{CR} .

□

Lemma 13 (Class table satisfies class requirements). *Given class declarations \bar{L} , such that $CT = \bigcup_{L' \in \bar{L}} (\text{addExt}(L') \cup \text{addCtor}(L') \cup \text{addFs}(L') \cup \text{addMs}(L'))$, a set of requirements CR , $\biguplus_{L' \in \bar{L}} (\text{removeMs}(CR, L') \uplus \text{removeFs}(CR, L') \uplus \text{removeCtor}(CR, L') \uplus \text{removeExt}(CR, L')) = CR'|_S$, and a substitution σ , such that $\sigma(CR)$ is ground, σ solves S , we have that if $\sigma(CR') = \emptyset$, then CT satisfies $\sigma(CR)$.*

Proof. By contradiction.

By assumption CT satisfies $\sigma(CR)$ does not hold. From this, follows $\exists(CReq, cond) \in CR$. $cond$ holds, i.e., all conditions of $cond$ do hold and no compatible clause with $CReq$ exists in CT .

As property of remove we add inequalities to invalidate requirements for which we know their exact types, as result exist at least one inequality condition that does not hold, and the requirement is removed, otherwise it remains in the requirements set.

First we consider the extend clauses in the requirements set. Let us consider $\exists(T.\text{extends}T', cond) \in CR$ such that $cond$ hold. By definition $\sigma(CR)$ is ground, namely $\sigma(T) = C', \sigma(T') = D'$. By assumption $(C'.\text{extends} : D') \notin CT$, i.e., the clause it is not member of any of the class declarations \bar{L} that are used to realize removing. Therefore after performing remove $\nexists \sigma(T) \neq C \in \sigma(cond')$ such that $\sigma(T) \neq C$ does not hold, where $(T.\text{extends} : T', cond') \in CR'$, i.e., $\sigma(cond')$ hold. Therefore $\sigma(CR') \neq \emptyset$.

This strategy of proof is used for constructor since from the definition of removeCtor only inequality conditions are added, and not considered while removing extends clause.

Second we consider field clauses. From the definition of removeFs and removeExt for every field clause we have a duplicated requirement corresponding to the parents type. Let us consider $\exists(T.f : T_f, cond) \in CR$. $cond$ hold. By definition $\sigma(CR)$ is ground, namely $\sigma(T) = C', \sigma(T_f) = D'$. By assumption $\nexists(D.f : D') \in CT$, such that $\sigma(T) <: D$. This means that in the class table was not added any *field clause*, such that f is declared in C' or in its parents. Therefore after performing remove $(T.f : T_f, cond') \in CR'$ we have that $\nexists(\sigma(T) \neq C) \in \sigma(cond')$. $(\sigma(T) \neq C)$ does not hold. i.e., $\sigma(cond')$ hold. Therefore $\sigma(CR) \neq \emptyset$.

The same strategy of proving is used for methods.

□

B.2. Equivalence of Contextual and Co-Contextual FJ

Theorem 17 (Equivalence for programs: \Leftarrow). *Given \bar{L} , if $\bar{L} \text{ OK} \mid S$, $\text{solve}(\Sigma, S) = \sigma$, where $\text{projExt}(\bar{L}) = \Sigma$, and σ is a ground solution, then $\bar{L} \text{ OK}$ holds.*

Proof. By induction on the typing judgment.

Case TC-PROGRAM with $\bar{L} \text{ OK} \mid S$.

Let $S = \bar{S} \cup S_{cr} \cup S_r$, σ is ground solutions and $\text{solve}(\text{projExt}(\bar{L}), S) = \sigma$, i.e., σ solves \bar{S} , S_{cr} , S_r .

By inversion, $\bar{L} \text{ OK} \mid \bar{S} \mid \overline{CR}$, $\forall i \in 1 \dots n$. $\sigma(CR_i)$ are ground.

$CR|_{S_c} = \text{merge}_{CR}(CR_1, \dots, CR_n)$.

$\uplus_{L' \in \bar{L}}(\text{removeMs}(CR, L') \uplus \text{removeFs}(CR, L') \uplus \text{removeCtor}(CR, L') \uplus \text{removeExt}(CR, L')) = CR_f|_{S_r}$, and $\sigma(CR_f) = \emptyset$

By Theorem 15, $CT \mid \bar{L} \text{ OK}$, and the correspondence relation holds, i.e., $\forall i \in [1..n]$. CT satisfies $\sigma(CR_i)$. CT satisfies $\sigma(CR_1) \cup \dots \cup \sigma(CR_n)$, i.e., CT satisfies $\sigma(CR)$ by Lemma 4.

Class table construction CT is $CT = \bigcup_{L' \in \bar{L}}(\text{addExt}(L') \cup \text{addCtor}(L') \cup \text{addFs}(L') \cup \text{addMs}(L'))$ by Lemma 13.

Then $\bar{L} \text{ OK}$ holds by rule T-PROGRAM . □